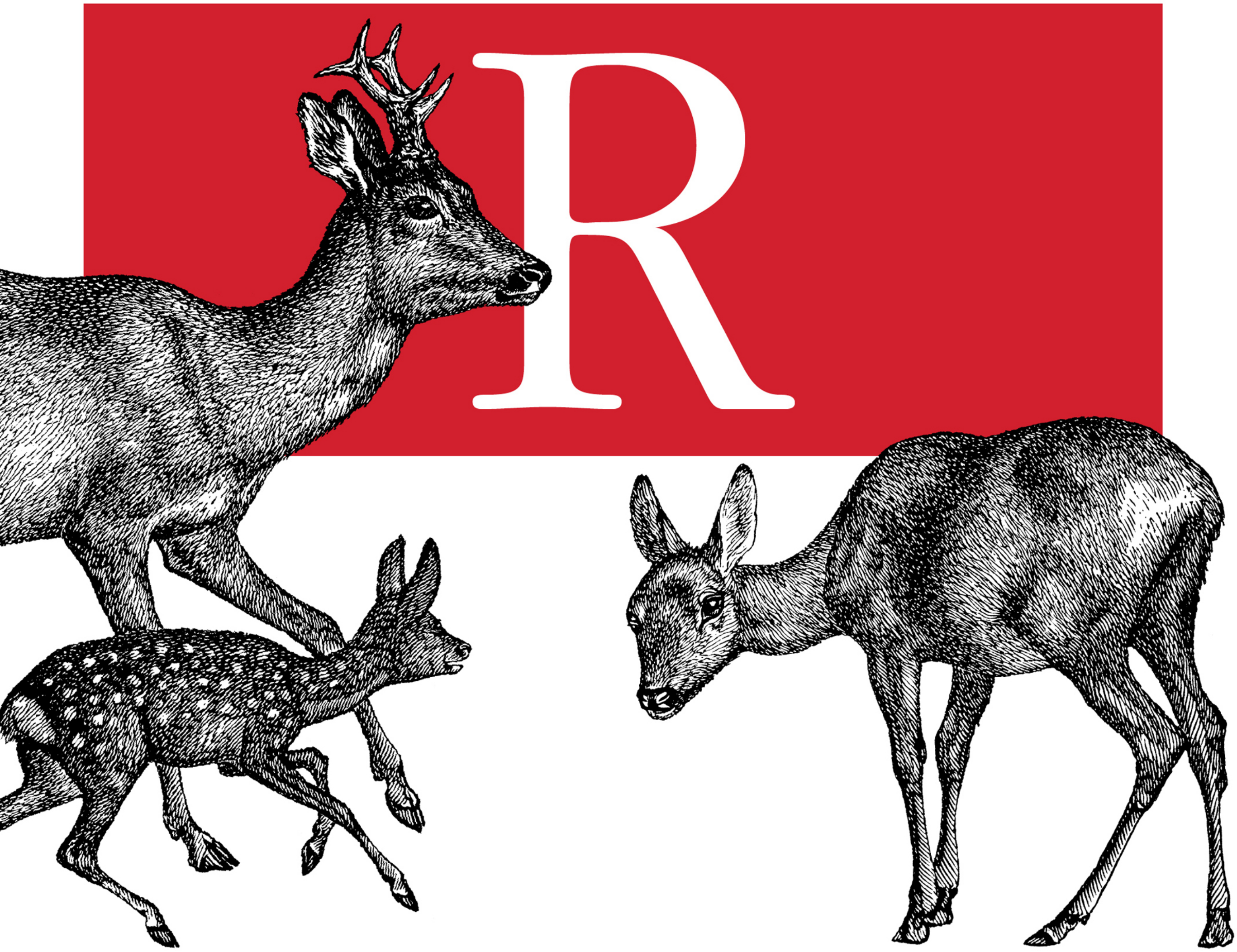
图灵程序设计丛书

Learning R

学习R



[美] *Richard Cotton* 著

刘军 译

O'REILLY®

人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

译者介绍

刘军

爱立信研发项目经理，2005年起从事无线网络嵌入式软件开发，后转入项目管理。爱技术，爱管理，也热衷于各种数据分析，Excel/matlab/origin玩得烂熟，最近开始迷上Hadoop和R。

图灵程序设计丛书

学习R

Learning R

[美] Richard Cotton 著
刘军 译

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo
O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北　　京

图书在版编目（C I P）数据

学习R / (美) 柯顿 (Cotton,R.) 著 ; 刘军译. --
北京 : 人民邮电出版社, 2014.6
（图灵程序设计丛书）
ISBN 978-7-115-35170-8

Ⅰ. ①学… Ⅱ. ①柯… ②刘… Ⅲ. ①程序语言－程
序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2014)第062931号

内 容 提 要

本书分为上下两部分，旨在指导你如何使用 R，并提供练习的机会。上半部分主要介绍 R 的技术细节和使用技巧。每章都简要介绍了一组不同的数据类型（例如第 4 章介绍向量、矩阵和数组）或概念（例如第 8 章介绍分支和循环）。下半部分更侧重实践，展示了从输入数据到发布结果这一标准的数据分析流程。

即使你没有任何编程基础，也能顺利阅读本书。

◆ 著　　　　[美] Richard Cotton
译　　　　刘　军
责任编辑　李松峰
执行编辑　李　静　郎婷婷
责任印制　焦志炜

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　　　印刷

◆ 开本：800×1000　1/16
印张：22.5
字数：532千字　　　　2014年 6 月第 1 版
印数：1－4 000册　　　2014年 6 月北京第 1次印刷
著作权合同登记号　图字：01-2014-2005号

定价：69.00元
读者服务热线：(010)51095186转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京崇工商广字第 0021 号

O'Reilly Media, Inc.介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O'Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项 O'Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar 博客有口皆碑。”

——*Wired*

“O'Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去 Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

版权声明

目录

第一部分　R 语言

第二部分 数据分析工作流

第三部分 附录

译者序

2011 年由布拉德 · 皮特主演的影片《点球成金》描述了一个运用数据运营球队的故事：一位落魄的棒球队总经理比利 · 比恩（Billy Beane）与他的 MBA 助理拍档在球队遭遇运营危机时，利用反传统的数据分析方法，开发出特有的计算机程序分析和选择球员、战术及其组合，使得这只球队从谷底反弹，而且创下美国棒球史上的连胜记录。他们不是只靠花大价钱购买高价球星，而是通过这种方法成功地使球队咸鱼翻生——“从没有一部电影将枯燥的数据转化为如此令人愉悦的娱乐体验”，《华尔街日报》如此评价。

上面只是数据分析中的一个有趣的例子，从中也反映出数据（以及大数据）分析已经逐渐成为一门越来越受关注的学问。在中国，人们同样越来越开始重视隐藏在数据背后的强大逻辑及其商业和学术价值。在我们的生活中，从“双 11”的购物狂欢，身边的饭团中出现的各种“股神”，工作中的项目管理，一直到家庭账目的管理，可以说数据分析无处不在，也越来越受到个人和企业的重视，已经有大量的数据科学家在为各大、中、小企业服务。

如何从数据中得到有趣和有价值的东西，本书正是通往这一目标的桥梁之一。本书详细介绍的开源语言 R 正日益成熟并受到关注，其社区和影响力也越来越大。

另外，和《华尔街日报》对以上电影的评价类似，这本书也将枯燥的数据和编程理论在某种程度上转化为更令人容易接受的体验——你可以在本书中接触到奥巴马与麦凯恩的选票大战，一只无辜的螃蟹的海底之旅，麋鹿的头盖骨，古英格兰七王国等演化过来的例子。通过这些既有趣又古怪的案例来学习 R 以及数据分析，难道不会更加刺激你的脑细胞从而加深记忆吗？

翻译的工作是辛苦的，且需要极度的耐心和细致，非常感谢图灵公司各位老师对我工作的支持和细心的审校。最后也非常感谢我的妻子潘玮倩在翻译时对我的理解和支持。

作为译者，由于水平和视野有限，翻译不当和错误之处再所难免。欢迎大家斧正，并提出宝贵建议。希望我们所做的工作能对你有帮助。祝阅读愉快，学有所成！

刘　军

2014 年 2 月

前言

R 是一种编程语言，也是用于数据分析和统计的软件环境。它是一个 GNU 项目，这意味着它是自由的开源软件。它正在以指数级的速度不断成长——普遍认为，它的用户人数可能超过了 100 万，它有 4000 多个由开发社区贡献的附件包，而且每年以约 25% 的速度增加。在本书创作之时，它在 Tiobe 编程社区指数（Tiobe Programming Community Index）的开发语言流行榜上已排至第 24 位，大致与 SAS 和 MATLAB 看齐。

R 广泛地应用在每一个需要统计或数据分析的领域，涵盖了金融、市场营销、医药、基因组学、流行病学、社会科学、教学以及许多其他较小的领域。

关于本书

因为 R 主要用于统计分析，所以很多关于 R 的书都在指导你如何计算统计或模型数据集。然而，这些书忽视了数据分析应用的实际情况。事实上，除非你做的是尖端研究，否则你所用到的统计技术往往只需用于常规任务，而且你的模型可能也不大。完整的数据分析流程更像是这样：

(1) 取得一些数据；
(2) 清理数据；
(3) 探索和可视化数据；
(4) 数据建模并做出预测；
(5) 展示或发布你的结果。

当然，每个阶段都可能碰到一些有趣的问题，以至于你需要更多的数据，或者要以不同的方式处理现有数据，这会使你的工作倒退一步。工作流是可以迭代的，但每个步骤都不可或缺。

本书的第一部分会从头开始教你 R——你不需要任何编程语言的经验。实际上，虽然完全

没有编程经验也无妨，但有一些基本的编程知识会更好。例如，本书介绍了如何注释代码以及编写 `for` 循环，但没有作更详细的解释。因此，如果你想要找本真正的编程入门课本，那么 Jason R. Briggs 写的 *Python for Kids* 非常合适！

本书的第二部分将展示 R 语言的完整数据分析流程，这里需要一些基本的统计知识。例如，你应该了解平均值和标准差等术语，以及什么是条形图（bar chart）。

本书最后将介绍 R 的一些高级主题，例如面向对象编程和包的创建。Garrett Grolem 的 *Data Analysis with R* 将会在本书的基础上深入探讨数据分析流程。

一点提醒：这不是一本参考书，许多主题叙述得并不详细。本书旨在指导你如何使用 R，并提供练习的机会。显然，我们没有那么多篇幅把所有 4000 个附件包都过一遍，但当读完此书，你将有能力找到你所需要的东西，并知道如何寻求帮助以应用它们。

本书主要内容

本书分为上下部分。上半部分旨在为你讲解 R 技术细节和使用技巧。每章都简要地介绍了一组不同的数据类型（例如第 4 章介绍向量、矩阵和数组）或概念（例如第 8 章介绍分支和循环）。

下半部分会更有意思：你能看到真实的数据分析。每章分别介绍标准数据分析流程的一部分，从输入数据到发布结果。

第一部分包括如下内容。

- 第 1 章：如何安装 R 以及在哪里得到帮助。
- 第 2 章：如何将 R 作为一个科学计算器使用。
- 第 3 章：以不同的方式检查变量。
- 第 4 章：向量、矩阵和数组。
- 第 5 章：列表和数据框（类似电子表格的数据）。
- 第 6 章：环境和函数。
- 第 7 章：字符串和因子（用于类别数据）。
- 第 8 章：分支（`if` 与 `else`）和基本的循环。
- 第 9 章：使用 `apply` 函数的高级循环应用函数及其变种。
- 第 10 章：如何安装和使用附件包。
- 第 11 章：日期和时间。

第二部分的主题如下。

- 第 12 章：如何将数据导入 R。

- 第 13 章：清理和操作数据。
- 第 14 章：通过统计计算和绘图探索数据。
- 第 15 章：如何建模。
- 第 16 章：各种高级编程技术。
- 第 17 章：如何为他人打包。

最后，第三部分的附录介绍一些有用的参考资料。

- 附录 A：比较不同类型变量的属性表。
- 附录 B：可以用 R 做的其他事。
- 附录 C：每章后测验题的答案。
- 附录 D：每章后编程练习题的答案。

你应该读哪几章

如果你从未使用过 R，那么请从第 1 章开始逐章阅读。如果你已经有了一些 R 的经验，不妨跳过第 1 章，快速浏览关于 R 语言的核心章节。

尽管章节间有一些关联，但因为每章都涉及一个不同的主题，所以你可以随机挑选感兴趣的章节阅读。

最近，我与 *R For Dummies* 的作者 Andrie de Vries 谈及这个问题，他竟然建议放弃此书，转而阅读他的著作[1]!

本书排版约定

本书中使用以下排版约定。

- 楷体
  表示新的术语。

- 等宽字体
  用于需逐字复制的代码示例，以及段内引用的编程元素，如变量或函数名、数据类型、环境变量、语句（statement）和关键字。代码块的输出也使用等宽字体，前面加上两个井号（##）。

- 斜等宽字体
  显示将由用户提供的值（或由上下文确定的值）替换的文本。

注 1：Andrie 的书所涵盖的内容和本书基本一致，且在许多方面和本书一样好。如果你也想阅读它，我对此表示欢迎。

本书中所用代码的风格指南：http://4dpiecharts.com/r-code-style-guide。

此图标表示小技巧、建议或一般说明。

此图标表示警告或小心。

目标、小结、小测验和练习

每章开头列出的目标会告诉你本章的内容概要，结束时的小结会重述已学的知识点，并会有个小测验以确保你是在集中精力地学习（而不是在看电视时假装看书）。测验答案可在本章内容中找到（或见本书的结尾部分，如果你想偷看的话）。每章最后还会有一些练习，大多是要求你写一些 R 的代码。每个练习题后面的方括号中有一个数字，它表示完成此题所需的大致时间。

使用代码

补充材料（代码范例、练习等）可以在此下载：http://cran.r-project.org/web/packages/learningr。

本书就是要帮读者解决实际问题的。也许你需要在自己的程序或文档中用到本书中的代码。除非大段大段地使用，否则不必与我们联系取得授权。因此，用本书中的几段代码写成一个程序不用向我们申请许可。但是销售或者分发 O'Reilly 图书随附的代码光盘则必须事先获得授权。引用书中的代码来回答问题也无需我们授权。将大段的示例代码整合到你自己的产品文档中则必须经过许可。

使用我们的代码时，希望你能标明它的出处。出处一般要包含书名、作者、出版商和书号，例如：*Learning R* by Richard Cotton (O'Reilly). Copyright 2013 Richard Cotton, 978-1-449-35710-8。

如果还有其他使用代码的情形需要与我们沟通，可以随时与我们联系：permissions@oreilly.com。

Safari® Books Online

Safari Books Online（www.safaribooksonline.com）是应需而变的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。

Safari Books Online 是技术专家、软件开发人员、Web 设计师、商务人士和创意人士开展调研、解决问题、学习和认证培训的第一手资料。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

http://oreil.ly.learningR

对于本书的评论和技术性问题，请发送电子邮件到：

bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：

http://facebook.com/oreilly

请关注我们的 Twitter 动态：

http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：

http://www.youtube.com/oreillymedia

致谢

本书优秀的编辑 Meghan Blanchette 给我提出了许多明智的建议。此外，很多才华横溢的人士为此书的完成提供了帮助。

数据由以下几位杰出的专业人士贡献。

- AMD 的 Bill Hogan 找到并清理了 Alpe d'Huez 循环数据集，还帮助我找到了 CDC 的淋病数据集。女士们，他特别要我强调他很健康。
- CEFAS 的 Ewan Hunter 提供北海蟹的数据集。
- 剑桥大学的 Corina Logan 整理并提供了鹿头骨的数据。
- 莱顿大学的 Edwin Thoen 整理并提供了奥巴马对战麦凯恩的数据集。
- Gwern Branwen 观看和阅读了数量惊人的日本动漫并整理了混血儿（hafu）数据集。赞一个！

还有另外很多人也为我提供了数据集，由于篇幅有限不能一一列出，在此表示感谢！

Bill Hogan、Marin 软件的 Daisy Vincent 和 JD Long 审阅了本书。我不知道 JD 在哪里工作，但他住在百慕大，有可能在三角带吧。其他提供意见和反馈的人包括：James White、Ben Hanks、Beccy Smith、TDX 集团的 Guy Bourne；HSL 的 Alex Hogg 和 Adrian Kelsey；Tom Hull、Karen Vanstaen、Rachel Beckett、Georgina Rimmer、Ruth Wortham、Bernardo Garcia-Carreras 以及 CEFAS 的 Joana Silva；特拉维夫大学的 Tal Galili；RStudio 的 Garrett Grolemund；纽约市立大学的 John Verzani。CEFAS 的 David Maxwell 出色地召集了几乎所有的 CEFAS 人士审阅了我的书。

非常感谢 John Verzani 帮助酝酿本书，并提供了结构方面的意见。

当我在为手稿的字符编码抓狂时，O'Reilly 的 Sanders Kleinfeld 提供了重要的技术支持。Yihui Xie 完全义务地帮我使用 knitr 生成 AsciiDoc。Rachel Head 在审稿时单枪匹马地发现了 4000 多个 bug、排印错误和其他错误。

Garib Murshudov 是第一个教我 R 的讲师，那得追溯到 2004 年。

最后，非常感谢 Janette Bowler 在我写作过程中给予的无比耐心和支持。

第一部分

R语言

第 1 章

简介

恭喜你踏上了 R 的编程之旅！现在，无需紧张，本章会让你很好地热身一下。在准备编码之前，我们先谈谈 R 是什么，以及如何安装并使用它。然后，尝试写一个程序，并学习如何获得帮助。

1.1 本章目标

读完本章后，你会了解以下内容：

- 可以用 R 做什么；
- 如何安装 R 和使用 IDE 工作；
- 写一个简单的 R 程序；
- 如何获取 R 的帮助。

1.2 R是什么

R 有时会以两种不同的面孔出现：R 编程语言及运行 R 程序的软件。乍一看可能会让人难以辨认，不过，大多数情况下都能够通过上下文，清楚地知道其中的 R 指的是什么。

R 语言由奥克兰大学的 Ross Ihaka 和 Robert Gentleman 在 20 世纪 90 年代初开发。它源于 20 世纪 70 年代 John Chambers 在贝尔实验室带头开发的 S 语言。R 软件是一个 GNU 项目，这表明它是一个重要的自由开源软件。现在的 R 语言和软件都由一个（目前）20 人的 R 核心团队（R Core Team）开发。

R 的历史可追溯到上世纪 70 年代，这一点非常重要，因为它在过去的几十年里不断演化，仿佛生命体一般，在演化过程中曾出现各种奇怪和矛盾的地方。与之相比，微软的 .NET Framework 更有一种“横空出世”[1] 的感觉。R 这种形式更为自由（尤其是自由许可证）的好处是，如果你不喜欢现有 R 的做法，完全可以重写一个包，使它按照你的想法去做。现在已经有很多人这样做了，如今常见的问题不是“我能使用 R 做到这一点吗”，而是“我应该用这三种实现中的哪一种”。

R 是一种解释型语言（有时也称为脚本语言），这意味着代码在运行之前并不需要编译。作为一种高级语言，R 旨在更快捷、更强大地为你分析数据，你无需知道代码如何运行于计算机底层这样的细节。

R 支持混合型的编程范式。它的核心是一种命令式语言（写一个脚本来逐条执行计算命令），但它也支持面向对象编程（数据和函数都绑定在类的内部）和函数式编程（函数是第一类对象，你可以像对待其他任何变量一样对待它们，而且也可以递归调用它们）。这种混合式的编程风格意味着 R 能拥有类似其他语言的特性。大括号意味着你可以编写看起来像 C 语言的命令式代码（但第 2 章将介绍的 R 的矢量特征会使它只需要更少的循环）。如果使用引用类，就可以写出有点像 C# 或 Java 的面向对象代码。函数式编程结构是受了 Lisp 的启发（变量的作用域规则取自 Lisp 的方言 Scheme），不过括号更少。所有这些其实都是在暗示 R 遵循 Perl 的理念 Perl ethos（http://bit.ly/14826CF）：

> 我们可以用多种方法来实现。
>
> ——Larry Wall

1.3 安装R

如果你正在使用一台 Linux 机器，那么很可能你的包管理器里面已经安装了 R，是不是最新版本则不好说。对于其他人，如果想安装 R，必须首先访问网站 http://www.r-project.org。不要因为它的网站[2] 看上去略微有些古老，就怀疑 R 的质量。在页面底部的 Getting Started 窗体中，点击 download R 链接。

选择一个离你最近的镜像后，在页面顶部的 Download and Install R 窗体中选择一个适合你的操作系统的链接。选择之后，就会出现一两个链接，然后按照提示选择并开始下载。

如果你是一个 Windows 用户并且不喜欢多次点击，可以用以下快捷方式下载安装文件：http://<CRAN MIRROR>/bin/windows/base/release.htm。

注 1：精心设计的结果？

注 2：看一下因特网档案馆的时光回溯机，就会发现它的首页自 2004 年 5 月之后就没有太大变化。

1.4 选择一个IDE

在 Windows 或 Mac OS X 下使用 R，会有一个图形用户界面（GUI）可用，这包括一个命令行解释器、显示绘图和帮助页面的部分，以及一个基本的文本编辑器。利用这些默认工具就完全可以使用 R，但是为应对更加复杂的编码活动，至少要使用一个更加强大的文本编辑器。有无数的文本编辑器可供选择，如果你已经有一个最喜欢的，就看看它是否能使 R 的代码语法高亮。

如果你不是非用某款编辑器不可，那么建议你使用集成开发环境（IDE）以得到最好的体验。和单独使用某个文本编辑器不同的是，这样你能得到使用 GUI 时的所有便利，外加一个强大的编辑器，甚至还有可能集成了版本控制功能。

下面将介绍五种常用的 IDE（当然还有一些建议），还有很多种，这里没有列出，你值得花上一些时间，从中选出最满意的那一种[3]，毕竟，在接下来的几千个小时中，你都会用到它。

1.4.1 Emacs+ESS

虽然 Emacs 说它自己只是一个文本编辑器，但过去 36 年的发展（还在持续发展中），已经让它具备了超多功能。如果你是一个编程老手，可能对于是否使用它早已有了自己的看法。粉丝对它几乎无限的可定制性及原始的编辑能力无比喜爱。但有人则抱怨它把事情弄得过于复杂，组合键的大量使用会导致肌肉劳损。学习它很有难度，所以做好心理准备，你可能要用一两个月的时间来习惯它。它的另一大好处是，除了 R 之外，它也适用于编写其他多种语言的程序。最初的 Emacs 版本（像 R 一样）是一个 GNU 项目，你可以从这里下载：http://www.gnu.org/software/emacs/。

另外一个流行的分支是 XEmacs，可以在这里下载：http://www.xemacs.org/。

ESS（Emacs Speaks Statistics）是一个能协助编写 R 代码的 Emacs 的插件。其实它也能用于 S-Plus、SAS 及 Stata，所以你可以使用你喜欢的任何软件包（选择 R 吧）来编写统计代码。ESS 的作者中有 R 的核心团队成员，所以它能与 R 良好地集成。你可以通过 Emacs 包管理系统获得它，或者从以下地址下载：http://ess.r-project.org/ 。

适用人群：要用多种语言编程，想要一个最强大的编辑器，而且又不怕学习困难。

1.4.2 Eclipse/Architect

Eclipse 是在 Java 社区中广泛使用的一种跨平台 IDE。它和 Emacs 一样强大，其插件系统

注 3：不必将 R 的使用方式限制为一种。我对 IDE 挺忠诚，可还使用过 Eclipse+StatET、RStudio、Live-R、Tinn-R、Notepad++ 和 R GUI。试着找到适合你的。

使其高度可定制化。它学起来相对容易，但与 Emacs 可以使用大量键盘操作相比，它需要更多的鼠标操作。

Architect 是一个面向 R 的 Eclipse 变种，由统计咨询公司 Open Analytics 开发，包括用于与 R 整合的 StatET 插件，以及一个优于 R GUI 内置的调试程序的调试器。下载地址是：http://www.openanalytics.eu/downloads/architect。

另外，你也可以先从 http://eclipse.org 下载 Eclipse IDE 的标准版本，再使用其包管理器从 http://www.walware.de/goto/statet 下载 StatET 插件。

适用人群：要用多种语言编程又没有时间学习 Emacs，而且不介意它的安装文件有几百兆。

1.4.3　RStudio

RStudio IDE 只能用于 R 开发。这意味着你不能（方便地）用它来编写其他语言的程序，但能得到一些 R 特有的功能。例如绘画窗口比原来的 R GUI 要好，而且它能提供发布代码的工具。它的编辑器比 Emacs 或 Eclipse 简单，但基本够用且更易上手。使用 RStudio 的好处是可以通过浏览器远程执行：你可以先在功能强大的服务器上运行 R，然后从上网本（或智能手机）远程访问而不会损失计算能力。你可以从这里下载它：http://www.rstudio.org。

适用人群：主要写 R 代码，不需要高级编辑功能，希望快速上手或者能远程执行代码。

1.4.4　Revolution-R

Revolution-R 有两种版本：自由社区版和企业付费版。两者都与之前提过的 IDE 不同——Emacs、Eclipse 和 RStudio 是纯图形化的前端，可让你使用任何版本的 R，但 Revolution-R 中的 R 版本是自己定制的，一般是一个稳定版本，比最新版本早一到两个版本。它还有一些增强的特性，如支持大数据以及一些企业功能。可由此下载：http://www.revolutionanalytics.com/products/revolution-r.php。

适用人群：主要用 R 编写代码，工作与大数据相关或想要一份付费的支持合同，又或者需要一个特别稳定的 R 平台。

1.4.5　Live-R

Live-R 算是一个新秀，截至本书出版时，它还只处于邀请测试的 beta 阶段。它为 R 提供了一个基于 Web 的 IDE，这样能避免安装软件的麻烦，而且它能像 RStudio 一样提供远程执行功能，让你能在一些动力不足的机器上运行 R 计算。Live-R 的协作功能包括共享的编辑器及共享的代码发布，以及一些基于 R 的运行课件管理工具。它的不足之处是，并非兼容所有的 R 扩展包，目前只限于大约 200 个与 Web 应用相兼容的扩展。你可由此注册：http://live-analytics.com/。

适用人群：主要使用 R 编写代码，不需要安装任何软件，或要讲授基于 R 的课程。

1.4.6 其他IDE和编辑器

你还可以使用很多其他的编辑器来编写 R 代码。例如：

- JGR（http://rforge.net/JGR，读作“Jaguar”）是一个基于 Java 的 GUI，它的特点是 GUI 有所加强；
- Tinn-R（http://www.sciviews.org/Tinn-R）是 TINN 编程器的一个分支，其扩展程序能帮助你编写 R 代码；
- SciViews-K（http://www.sciviews.org/SciViews-K），由创建 Tinn-R 的相同队伍开发，是一个用于 R 开发的 Komodo IDE 的插件；
- Vim-R（http://www.vim.org/scripts/script.php?script_id=2628）是一个用于整合 R 的 Vim 插件；
- NppToR（http://sourceforge.net/projects/npptor）是一个 Notepad++ 的 R 语言插件。

1.5 你的第一个程序

在几乎所有的编程书籍中，第一个例子都似乎应该是一个输出“Hello world！”的程序。但对 R 来说，这很无聊，因为你只需要在命令提示符下输入“Hello world！”就行了，然后它就会打印相同内容。所以，我们还是写个简单的统计程序吧。

打开 R 的 GUI 程序或你决定使用的任何 IDE，找到命令提示符（在代码编辑器窗口），然后键入：

```
mean(1:5)
```

按下回车键运行代码行，你应该会得到答案 3。你可能已经猜到了，这个代码是计算从 1 到 5 的算术平均值。冒号运算符“ ：”在本例中会创建一个从第一个数字 (1) 到第二个数字 (5) 的序列，每个相隔为 1。计算得到的序列称为一个*矢量*。`mean` 是一个*函数*（计算算术平均值），在括号内的向量被称为函数的*参数*。

干得好！你已经使用 R 完成了一个统计任务。

> 在 R 的 GUI 和这里提到的大多数的 IDE 中，你可以按向上箭头键循环执行以前的命令。

1.6 如何从R中获得帮助

在开始写 R 代码之前，最重要的是要了解如何得到帮助。有多种方法可以得到帮助。首先，如果你想知道某个函数或数据集的信息，可以输入 ?，后面加上函数名。如果你想查找某个函数，输入两个问号 (??)，后面加上与此函数相关的关键词。对于特殊字符、关键字和多个字词的搜索需要加上单引号或双引号。例如：

```
?mean                      # 打开 mean 函数的帮助页面
?"+"                       # 打开加法操作的帮助页面
?"if"                      # 打开 if 的帮助页面，用于分支代码
??plotting                 # 搜索所有包含 "plotting" 的主题
??"regression model"       # 搜索所有与 regression model 相关的主题
```

符号表示注释。这意味着 R 将忽略此行的其他部分。使用注释来为你的代码添加说明，这样就可以使你记起以前做过的事。

函数 help 及 help.search 分别等同于 ? 及 ??，但是你必须把你的参数括在引号中。以下命令与之前的相当：

```
help("mean")
help("+")
help("if")
help.search("plotting")
help.search("regression model")
```

apropos 函数[4] 能找到匹配其输入的变量（以及函数）。如果你能记住部分已创建的变量或要使用的函数名，apropos 就会非常好用。例如，假设你已经创建了一个变量 a_vector：

```
a_vector <- c(1, 3, 6, 10)
```

你可以通过 apropos 重新记起这个变量：

```
apropos("vector")

## [1] ".__C__vector"     "a_vector"              "as.data.frame.vector"
## [4] "as.vector"        "as.vector.factor"      "is.vector"
## [7] "vector"           "Vectorize"
```

结果包含你刚刚创建的变量 a_vector 以及所有其他包含 vector 字符串的变量。在这个例子中，其他的函数都是 R 的内置函数。

找到包含特定字符串的变量固然是好，但你也可以把 apropos 结合正则表达式来更精确地匹配。

注 4：拉丁文，意为“一个能搜索联机手册的 UNIX 程序”。

正则表达式是匹配字符串的一个跨语言的语法。本书只会稍作涉猎，但是建议你学会使用它们，因为它能改变你的生活。从 http://www.regular-expressions.info/quickstart.html 开始，然后看看 Michael Fitzgerald 的《学习正则表达式》吧。

例如，apropos 的以下简单示例试图寻找所有以 z 结尾的变量，或者含有 4 到 9 之间数字的所有变量：

```
apropos("z$")

## [1] "alpe_d_huez" "alpe_d_huez" "force_tz"  "indexTZ"   "SSgompertz"
## [6] "toeplitz"    "tz"          "unz"       "with_tz"

apropos("[4-9]")

##  [1] ".__C__S4"             ".__T__xmlToS4:XML"   ".parseISO8601"
##  [4] ".SQL92Keywords"       ".TAOCP1997init"      "asS4"
##  [7] "assert_is_64_bit_os" "assert_is_S4"        "base64"
## [10] "base64Decode"         "base64Encode"        "blues9"
## [13] "car90"                "enc2utf8"            "fixPre1.8"
## [16] "Harman74.cor"         "intToUtf8"           "is_64_bit_os"
## [19] "is_S4"                "isS4"                "seemsS4Object"
## [22] "state.x77"            "to.minutes15"        "to.minutes5"
## [25] "utf8ToInt"            "xmlToS4"
```

大多数函数都能通过查找相关的范例来更好地了解它们的工作原理。你可以使用 example 函数查看它们。也有一些较长的概念演示，你可以通过 demo 函数查看：

```
example(plot)
demo()          # 列出所有演示
demo(Japanese)
```

R 是模块化的，它被分成不同的包（后面将详细讨论），其中一些包含*片段*（vignettes），是指导如何使用这些包文件的短文档。可以使用 browseVignettes 来浏览所有在你机器上的片段：

```
browseVignettes()
```

你还可以使用 vignette 函数访问一个特定的片断（但如果你的记性和我一样糟糕，与其尝试记住某个包和片断的名称，还不如结合 browseVignettes 和网页搜索）：

```
vignette("Sweave", package = "utils")
```

帮助搜索操作符 ?? 和 browseVignettes 只会发现那些你已经安装了的包里的东西。如果你想查找“任何”包，可以使用 RSiteSearch，它会查询整个 http://search.r-project.org 网站的包。多个单词组成的短语必须用大括号括上：

```
RSiteSearch("{Bayesian regression}")
```

学习如何自我帮助非常重要。想出一个与你工作相关的保留字，然后尝试使用 `?`、`??`、`apropos` 和 `RSiteSearch` 搜索它吧。

互联网上有大量与 R 相关的资源。你可以从以下这些开始。

- R 有一些邮件列表（http://www.r-project.org/mail.html），收集了多年来积累的关于语言的各种问题。最起码值得订阅通用列表 R-help。
- RSeek（http://rseek.org）是一个 R 的网页搜索引擎，能查找出各种函数、R 邮件列表归档中的讨论和博客文章。
- R- 博客（http://www.r-bloggers.com）是 R 主要的博客社区，这是关注 R 社区新闻和小技巧的最佳方式。
- 编程问答网站 Stack Overflow（http://www.stackoverflow.com），也是一个活跃的 R 社区，它提供了一个可替代的 R-help 邮件列表。你还可以通过回答问题得到点数和徽章！

1.7 安装其他相关软件

还有其他几个软件可以扩展 R 的功能。在 Linux 下，你的包管理器应该能够把它们检索出来。而在 Windows 环境下，与其在互联网四处查找，你还不如通过 `installr` 插件包来自动安装这些额外的软件。这些软件并非必不可少，所以你愿意的话可以跳过这一节，但起码你应该知道它们的存在，以备不时之需。如果你还不能理解这些安装和装载包的命令，不必担心，因为第 10 章会详细讨论它们：

```
install.packages("installr")   # 下载并安装 installr 包
library(installr)              # 装载 installr 包
install.RStudio()              # 下载并安装 RStudio IDE
install.Rtools()               # 你需要 Rtools 来构建自己的包
install.git()                  #git 提供了代码的版本控制功能
```

1.8 小结

- R 是一个自由的开源数据分析语言。
- 它也是一个用于运行在 R 程序中的软件。
- 从 http://www.r-project.org 下载 R。
- 可以使用任何文本编辑器写 R 代码，但也有几个集成开发环境能使开发更容易。
- 键入 `?` 加上函数名字来获得帮助。
- 输入 `??` 加上字符串找到有用的功能，或调用 `apropos` 函数。
- 网上有很多 R 的资源。

1.9 知识测试：问题

- 问题 1-1
  R 是哪种语言开源版本？

- 问题 1-2
  说出至少两种 R 的编程模式。

- 问题 1-3
  用什么命令可以创建一个从数 8 到 27 的矢量？

- 问题 1-4
  在 R 中用于搜索帮助的函数是哪个？

- 问题 1-5
  在互联网上用于搜索 R 的相关帮助的函数是哪个？

1.10 知识测试：练习

- 练习 1-1
  访问 http://www.r-project.org，下载并安装 R。另外，下载并安装 1.4.6 节中提到的 IDE。[30]

- 练习 1-2
  `sd` 函数会计算标准差。请算出从数 0 到 100 的标准差。
  提示：答案应该是 `29.3`。[5]

- 练习 1-3
  观看数学符号演示，数学符号使用 `demo(plotmath)`。[5]

第 2 章

科学计算器

本质上，R 是一个强大的科学计算器，因此它有一套相当全面的内置数学功能。本章将介绍算术运算符、常用的数学函数以及关系运算符，并告诉你如何为变量赋值。

2.1 本章目标

阅读本章后，你会了解以下内容：

- 如何把 R 当成一个科学计算器来使用；
- 如何给变量赋值并查看它的值；
- 如何使用无限值和缺失值（missing value）；
- 什么是逻辑向量以及如何操作它们。

2.2 数学运算符和向量

运算符 + 执行加法，它还有一个特殊技巧：除了把两个数字相加之外，还可把两个向量相加。向量是数值的有序集，它在统计学中极其重要，因为通常的分析对象是整个数据集，而不仅是一条数据。

在上一章你已经看到，冒号运算符 : 能创建一个从某个数值开始到另一个数值结束的序列。而 c 函数则会把一系列的值拼接起来创建向量（这里 c 是 concatenate 的第一个字母，concatenate 是一个拉丁词，意思是“把所有东西连接在一起”）。

R 中的变量名是区分大小写的，因此下例中我们要留心一点。大写的 C 函数与小写的 c 函

数作用完全不同[1]：

```
1:5 + 6:10         #look, no loops!
## [1]  7  9 11 13 15
c(1, 3, 6, 10, 15) + c(0, 1, 3, 6, 10)
## [1]  1  4  9 16 25
```

冒号运算符和 c 函数在 R 代码中几乎无处不在，好好练习使用它们吧。现在，试试创建你自己的向量。

如果要用 C 或 Fortran 语言编写代码，我们需要编写一个循环语句来为向量中的每个元素执行加法。R 的向量化加法操作符把事情简单化了，使我们无需使用循环语句。2.5 节将对向量作更多的介绍。

在 R 中，向量化有几种含义，其中最常见的含义是：运算符或函数能作用于向量中的每个元素，而无需显式地编写循环语句（这种内置的基于元素的隐式循环也远远快于显式地写循环语句）。向量化的第二个含义是：当一个函数把一个向量作为输入时，能计算汇总统计：

```
sum(1:5)

## [1] 15

median(1:5)

## [1] 3
```

向量化的第三种含义不太常见，即参数的向量化，例如当函数根据输入参数计算汇总统计时。sum 函数就是这样，不过这非常特殊。而 median 函数则不是这样：

```
sum(1, 2, 3, 4, 5)

## [1] 15

median(1, 2, 3, 4, 5) # 这会抛出错误

## Error: unused arguments (3, 4, 5)
```

在 R 中，不只是加号（+），其他所有算术运算符都是向量化的。下例将演示减法、乘法、幂运算、两种除法及余数。

```
c(2, 3, 5, 7, 11, 13) - 2            # 减法

## [1]  0  1  3  5  9 11

-2:2 * -2:2                          # 乘法
```

注 1：有一些其他的名称冲突：filter 和 Filter，find 和 Find，gamma 和 Gamma，nrow/ncol 和 NROW/NCOL。因为 R 是一种不断演进而非一蹴而就的语言，这是一个令人遗憾的副作用。

```
## [1] 4 1 0 1 4

identical(2 ^ 3, 2 ** 3)             # 我们可用 ^ 或 ** 代表求幂
                                     # 尽管用 ^ 更加普遍

## [1] TRUE

1:10 / 3                             # 浮点数除法

## [1] 0.3333 0.6667 1.0000 1.3333 1.6667 2.0000 2.3333 2.6667 3.0000 3.3333

1:10 %/% 3                           # 整数除法

## [1] 0 0 1 1 1 2 2 2 3 3

1:10 %% 3                            # 余数

## [1] 1 2 0 1 2 0 1 2 0 1
```

R 还包含了多种数学函数。有三角函数（sin、cos、tan，以及相反的 asin、acos 和 atan）、对数和指数（log 和 exp，以及它们的变种 log1p 和 expm1，这两个函数对那些非常小的 x 值能更加精确地计算 log(1 + x) 和 exp(x - 1) 的值），以及几乎所有其他你能想到的数学函数。请参考下例并加以理解。请再次注意，所有的函数都作用于向量，而不仅仅是单个值。

```
cos(c(0, pi / 4, pi / 2, pi))        #pi 是内置常数

## [1] 1.000e+00  7.071e-01  6.123e-17 -1.000e+00

exp(pi * 1i) + 1                     # 欧拉公式

## [1] 0+1.225e-16i

factorial(7) + factorial(1) - 71 ^ 2 #5041 是一个大数字

## [1] 0

choose(5, 0:5)

## [1]  1  5 10 10  5  1
```

要比较整数值是否相等请使用 == 而不是单个等号 =，因为我们将会看到，单个等号另有用途。正如算术运算符一样，== 和其他关系运算符都是向量化的。要检查是否不相等，“不等”运算符为 !=。你可能已经猜到大于和小于号就是 > 和 <（如果有可能相等，则使用 >= 和 <=）。以下是几个例子：

```
c(3, 4 - 1, 1 + 1 + 1) == 3          # 操作符也是向量化的

## [1] TRUE TRUE TRUE

1:3 != 3:1
```

```
## [1]  TRUE FALSE  TRUE
exp(1:5) < 100
## [1]  TRUE  TRUE  TRUE  TRUE FALSE
(1:5) ^ 2 >= 16
## [1] FALSE FALSE FALSE  TRUE  TRUE
```

使用 == 来比较非整型变量可能会带来问题。到目前为止。我们处理的所有数字都是浮点数。这意味着，对于两个数 a 和 b 来说，它们可存储为 a * 2 ^ b。由于它们都以 32 位存储，所以只能是一个近似值。这意味着舍入误差（rounding error）会常常潜伏在你的计算之中，你预期的答案可能是完全错误的。有很多书专门介绍这个主题，由于内容较多，这里就不过多介绍了。这是个常见的错误，R 中的 FAQ 上有一个关于它的条目（http://bit.ly/17jZFfE），便于你深入了解。

看以下两个数字，它们应该是相同的。

```
sqrt(2) ^ 2 == 2        #sqrt 是函数的平方根
## [1] FALSE
sqrt(2) ^ 2 - 2         # 这个微小的差值就是舍入误差
## [1] 4.441e-16
```

R 还提供了 all.equal 函数用于检查数字是否相等。它提供了一个容忍度（tolerance level，默认情况下为 1.5e-8），因而那些小于此容忍度的舍入误差将被忽略：

```
all.equal(sqrt(2) ^ 2, 2)
## [1] TRUE
```

如果要比较的值不一样，all.equal 返回时将报告其差值。如果你需要它们在比较后返回的是一个 TRUE 或 FALSE 值，则应把 all.equal 函数嵌入 isTRUE 函数中调用：

```
all.equal(sqrt(2) ^ 2, 3)
## [1] "Mean relative difference: 0.5"
isTRUE(all.equal(sqrt(2) ^ 2, 3))
## [1] FALSE
```

要检查两个数字是否一样，不要使用 ==，而使用 all.equal 函数。

我们也可以使用 `==` 来比较字符串。在这种情况下，比较会区分大小写，所以字符串必须完全匹配。理论上，也可以使用大于或小于（`>` 和 `<`）来比较字符串：

```
c(
  "Can", "you", "can", "a", "can", "as",
  "a", "canner", "can", "can", "a", "can?"
) == "can"

##  [1] FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE  TRUE  TRUE FALSE
## [12] FALSE

c("A", "B", "C", "D") < "C"

## [1]  TRUE  TRUE FALSE FALSE

c("a", "b", "c", "d") < "C"             # 你的结果可能有所不同

## [1]  TRUE  TRUE  TRUE FALSE
```

然而在实际中，后一种方式总相当蹩脚，因为结果取决于你的语言环境（不同文化中充满了奇怪的字母排序规则，例如在爱沙尼亚语中，“z”排在“s”和“t”之间）。13.2 节将讨论更强大的字符串匹配功能。

> 在帮助页面中 `?Arithmetic`、`?Trig`、`?Special` 和 `?Comparison` 有更多的例子，并提供边界情况下的大量处理细节。（如有兴趣，请试试 `0 ^ 0` 或使用整数除以非整数。）

2.3　变量赋值

虽然计算很美妙，但通常我们要存储结果以供重用。我们可以使用 `<-` 或 `=` 给（本地）变量赋值，虽然由于历史原因，`<-` 是首选。

```
x <- 1:5
y = 6:10
```

现在，我们可以重新使用这些值进行下一步计算：

```
x + 2 * y - 3

## [1] 10 13 16 19 22
```

请注意，在给变量 `x` 和 `y` 赋值之前，并不需要声明它们（这和大多数编译语言不一样）。事实上，我们不能声明任何类型，因为在 R 中不存在这种概念。

变量名可包含字母、数字、点和下划线，但它不能以数字或一个点后跟数字（因为看起来太像一个数字）开头。系统的保留字也是不允许的，如 if 和 for。某些语言环境（locale）下允许非 ASCII 字母，但是为了代码的可移植性，最好使用以 a 到 z（A 到 Z）的字母。

关于命名规则的细节，请参见帮助页面 ?make.names。

赋值运算符两边的空格并不是必须的，但它们有助于提高代码的可读性，尤其是对于 <- 来说，两边的空格可以轻松将它与小于号区分开：

```
x <- 3
x < -3
x<-3          # 这是赋值运算符还是小于号?
```

我们还可使用 <<- 来对全局变量赋值。对此，在 6.1 节谈到环境和范围时会再作深入探讨。现在，只需把它看做创建了一个可在任意地方使用的变量：

```
x <<- exp(exp(1))
```

另一个变量赋值的方法是通过 assign 函数赋值。这种情况不太常见，但在极个别情况下，使用函数语法来对变量赋值是很有用的。本地的（“标准的”）赋值函数需要两个参数：要赋值的变量名以及要赋予该变量的值。

```
assign("my_local_variable", 9 ^ 3 + 10 ^ 3)
```

全局赋值（如 <<- 操作符一样）还需要加上一个参数：

```
assign("my_global_variable", 1 ^ 3 + 12 ^ 3, globalenv())
```

目前，不用担心 globalenv 函数，第 6 章将作详细的解释。

相比 <- 操作符，使用 assign 函数会使代码可读性变差，因此须谨慎使用。有时，在一些涉及环境变量的高级程序设计中，它会使事情变得更简单。但如果你的代码中到处都是 assign 函数，可能就出错了。

还要注意的是，assign 函数不会检查第一个参数是否是一个有效的变量名，它只是创建它。

请注意，当你对一个变量赋值时，不会马上看到值是多少。要看到变量包含的值，只需在命令提示符下键入其名称即可将其打印出来。

```
x

## [1] 1 2 3 4 5
```

在某些系统中，例如从一个 Linux 终端运行 R 时，可能需要显式调用 print 函数以查看该值，即输入 print(x)。

如果你想把赋值和打印语句都写在同一行，有两种选择。第一，把多个语句放在一行，中间用分号 ; 分开。第二，把赋值语句写在括号 () 中。在下例中，rnorm 函数生成正态分布

的随机数，而 `rlnorm` 函数则生成对数正态分布的随机数[2]：

```
z <- rnorm(5); z

## [1] 1.8503 -0.5787 -1.4797 -0.1333 -0.2321

(zz <- rlnorm(5))

## [1] 1.0148 4.2476 0.3574 0.2421 0.3163
```

2.4 特殊数字

为了帮助算术运算，R 支持四种特殊值：`Inf`、`-Inf`、`NaN` 和 `NA`。显然，前两个分别是正负无穷，而后两个则需做些解释。`NaN` 为“不是一个数”（not-a-number）的缩写，它意味着我们的计算或没有数学意义，或无法正确执行。`NA` 是“不可用”（not available）的缩写，并代表缺失值——这个问题在数据分析中会经常碰到。在一般情况下，如果我们的计算涉及一个缺失值，则结果也将丢失。

```
c(Inf + 1, Inf - 1, Inf - Inf)

## [1] Inf Inf NaN

c(1 / Inf, Inf / 1, Inf / Inf)

## [1] 0 Inf NaN

c(sqrt(Inf), sin(Inf))

## Warning: NaNs produced

## [1] Inf NaN

c(log(Inf), log(Inf, base = Inf))

## Warning: NaNs produced

## [1] Inf NaN

c(NA + 1, NA * 5, NA + Inf)

## [1] NA NA NA
```

当算术中涉及 `NA` 和 `NaN` 时，得到的结果将为这两个值之一，取哪个值则取决于所使用的系统：

```
c(NA + NA, NaN + NaN, NaN + NA, NA + NaN)

## [1] NA NaN NaN NA
```

注 2：由于数字是随机产生的，所以你自己尝试时可能会得到不同的值。

可使用函数来检查这些特殊值。请注意，NaN 和 NA 既非有限值亦非无限值，NaN 代表缺失值，而 NA 是一个数字。

```
x <- c(0, Inf, -Inf, NaN, NA)

is.finite(x)

## [1]  TRUE FALSE FALSE FALSE FALSE

is.infinite(x)

## [1] FALSE  TRUE  TRUE FALSE FALSE

is.nan(x)

## [1] FALSE FALSE FALSE  TRUE FALSE

is.na(x)

## [1] FALSE FALSE FALSE  TRUE  TRUE
```

2.5 逻辑向量

除了数字以外，科学计算还经常涉及逻辑值，特别是当使用关系运算符（< 等）时。许多编程语言都使用布尔逻辑，其中的值可为 TRUE 或 FALSE。在 R 中，情况有点复杂，因为还可能有缺失值 NA。有时，这种拥有三种状态的系统被称为 troolean 逻辑，虽然这可算是一个词源学上的冷笑话，因为“Boolean”中的“Bool”源于 George Bool，而与二进制无关。

TRUE 和 FALSE 是 R 中的保留字：你不能创建以它们命名的变量（但可使用它们的小写或大小写混合，如 True）。当你启动 R 时，变量 T 和 F 已被系统默认定义为 TRUE 和 FALSE。虽然这能让你少打点字，但也会造成很大的问题。T 和 F 不是保留字，因此用户可以重新定义它们。这意味着你可以在命令行中使用它们的缩写名称，但如果你的代码需要与他人的代码交互（特别是当他们的代码涉及时间、温度或数学函数时），请避免使用这两个缩写。

在 R 中有三个向量化逻辑运算符：

- ! 代表非操作
- & 代表与操作
- | 代表或操作

```
(x <- 1:10 >= 5)

##  [1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE

!x

##  [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
(y <- 1:10 %% 2 == 0)

##  [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE

x & y

##  [1] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE  TRUE

x | y

##  [1] FALSE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

我们可编出一些真值表来看看它们是如何工作的（请不用管这段代码是否有意义，只需集中精力去理解每个值是如何在真值表中计算出来的）：

```
x <- c(TRUE, FALSE, NA)          # 三个逻辑值
xy <- expand.grid(x = x, y = x)  # 取得 x 和 y 的所有组合
within(                          # 在 xy 内赋值
  xy,
  {
    and <- x & y
    or <- x | y
    not.y <- !y
    not.x <- !x
    }
  )

##       x     y not.x not.y    or   and
## 1  TRUE  TRUE FALSE FALSE  TRUE  TRUE
## 2 FALSE  TRUE  TRUE FALSE  TRUE FALSE
## 3    NA  TRUE    NA FALSE  TRUE    NA
## 4  TRUE FALSE FALSE  TRUE  TRUE FALSE
## 5 FALSE FALSE  TRUE  TRUE FALSE FALSE
## 6    NA FALSE    NA  TRUE    NA FALSE
## 7  TRUE    NA FALSE    NA  TRUE    NA
## 8 FALSE    NA  TRUE    NA    NA FALSE
## 9    NA    NA    NA    NA    NA    NA
```

其他两个比较有用的处理逻辑向量的函数是 `any` 和 `all`，如果输入向量中至少包含一个 `TRUE` 值或只包含 `TRUE` 值，它们将分别返回为 `TRUE`：

```
none_true <- c(FALSE, FALSE, FALSE)
some_true <- c(FALSE, TRUE, FALSE)
all_true <- c(TRUE, TRUE, TRUE)
any(none_true)

## [1] FALSE

any(some_true)

## [1] TRUE
```

```
any(all_true)

## [1] TRUE

all(none_true)

## [1] FALSE

all(some_true)

## [1] FALSE

all(all_true)

## [1] TRUE
```

2.6 小结

- R 可用作一个非常强大的科学计算器。
- 给变量赋值以便重复使用。
- R 有正负无穷、无值可用、缺失值这几种特殊值以协助数学运算。
- R 使用 troolean 逻辑。

2.7 知识测试：问题

- 问题 2-1
  用于整数除法的操作符是什么？

- 问题 2-2
  如何检查变量 `x` 是否等于 `pi`？

- 问题 2-3
  至少描述两种给变量赋值的方式。

- 问题 2-4
  这 5 个数字中哪些是无限值：`0`，`Inf`，`-Inf`，`NaN` 和 `NA`？

- 问题 2-5
  5 个数字中哪些不是缺失值：`0`，`Inf`，`-Inf`，`NaN` 和 `NA`？

2.8 知识测试：练习

- 练习 2-1
  计算 1~1000 所有整数的倒数的反正切（即 arctan）。提示：参考 `?Trig` 帮助页面找到反

正切函数。你并不需要一个函数来计算倒数。[5]

给变量 `x` 分配从 1 到 1000 的数字向量。计算 `x` 的倒数的反正切值，如第 1 题所示，然后将其分配给变量 `y`。现在逆转此操作，计算 `y` 的切线的倒数，然后把值赋给变量 `z`。[5]

- 练习 2-2

  使用 `==` 符号、`identical` 和 `all.equal` 函数，比较练习 2-1 第 2 题中的 `x` 和 `z` 变量。对于 `all.equal`，尝试通过传入函数的第三个参数改变其容差度。如果容差设置为 `0`，会发生什么？ [10]

- 练习 2-3

  定义下面的向量。

  (1) 把 `true_and_missing` 赋值为 `TRUE` 和 `NA`（至少其中一个，不限顺序）。

  (2) 把 `false_and_missing` 赋值为 `FALSE` 和 `NA`。

  (3) 把 `mixed` 赋值为 `TRUE`、`FALSE` 和 `NA`。

  将 `any` 和 `all` 函数应用于以上每个向量。[5]

第3章

检查变量和工作区

到目前为止，我们已经进行了一些运算操作以及变量赋值。本章将探讨检查这些变量的属性的方法，并对具有这些属性的用户工作区进行操作。

3.1 本章目标

阅读本章后，你会了解以下内容：

- 什么是类，以及一些常见类的名称；
- 如何转换变量类型；
- 如何检查变量并查找有用的信息；
- 如何操作用户工作区。

3.2 类

R 中的所有变量都有一个类，表明此变量属于什么类型。例如，大部分的数字是 `numeric` 类（其他类型请参见下一节），逻辑值是 `logical` 类。其实，因为 R 没有标量类型（scalar type），所以更严格地说，数字向量应该是 `numeric` 类，逻辑值向量是 `logical` 类。在 R 中“最小的”数据类型是向量。

可使用 `class(my_variable)` 来找出变量的类名：

```
class(c(TRUE, FALSE))

## [1] "logical"
```

值得的注意是，所有的变量除了类之外，还有一个内部存储类型（通过 `typeof` 访问）、一

个模式（mode），以及一个存储模式（storage.mode）。这听起来很复杂，不过无需担心，因为类型、模式和存储模式大多为历史遗留存在，实际中你只需使用对象的类（除非你加入了 R 的核心团队）。附录 A 有一个参照表显示了各种变量类型的类、类型和（存储）模式及其关系。如果你不认识其中的一些类，不用担心，也没必要死记硬背。你只需浏览表格，注意它们之间的关联性。

简单起见，从现在开始，我会把“类”（class）和“类型”（type）完全等同起来（除非另作说明）。

3.3 不同类型的数字

我们在前一章所创建的所有变量都是数字，但 R 包含三种不同类别的数值变量：浮点值 numeric、整数 integer 和复数 complex。可通过检查变量的类型 class 把它们分辨出来：

```
class(sqrt(1:10))
## [1] "numeric"
class(3 + 1i)          #"i" 创建了复数的虚部
## [1] "complex"
class(1)               # 尽管只有一个数字，这也是一个 numeric 类
## [1] "numeric"
class(1L)              # 添加 L 后缀把数字变为整型
## [1] "integer"
class(0.5:4.5)         # 冒号操作符返回的值是 numeric 类……
## [1] "numeric"
class(1:5)             # 除非所有值都是整数
## [1] "integer"
```

请注意，当写作本书时，即使把 R 程序安装在 64 位操作系统上，所有的浮点数仍是 32 位的（“双精度”），而 16 位（“单精度”）的数字是不存在的。

输入 .Machine 将显示一些 R 的数字属性信息。虽然从理论上说，这些值在不同的机器上可能不一样，但在大多数的构建（build）中，大部分的值是相同的。通常情况下，你无需留意 .Machine 返回的值。但值得注意的是，R 中最大的全精度浮点数是 1.8e308。此数值对于满足日常工作已足够大，但比无穷大要小得多！可以表示的最小正数是 2.2e-308。最大的整数为 2 ^ 31 - 1，它不过比二十亿多一点儿罢了（反之最小的负整数为 -2 ^ 31 + 1）[1]。

注 1：如果这些极限值还不够你用，你可以从 Rmpfr 包中得到更高精度的值，或从 brobdingnab 包得到非常大的数字。不过，这些都是非常少见的需求，而 R 中的三个内置数字类几乎能适用于所有用途。

其他唯一值得注意的数值是 ε，它是最小的正浮点数，例如 `|ε + 1| != 1`。你可以用这种奇特的方法来表示两个数是如此之接近，因而 R 知道它们是不同的。它的值大概为 `2.2e-16`。在你使用 `all.equal` 来比较两个数字向量是否相等时，这个值就会被派上用场。

事实上，所有这些都比你想象的更简单，因为完全不可能（故意地）不使用整数。R 的设计使得几乎在所有地方都需要使用整数，例如对一个向量进行索引。浮点数也是如此。

3.4 其他通用类

除了我们已知的三个数字类和逻辑类，向量还有其他三个类，它们分别是：用于存储文本的字符 `character`，存储类别数据的因子 `factor`，以及较罕见的存储二进制数据的原始值 `raw`。

在下例中，正如之前创建数字向量一样，我们使用 `c` 运算符创建了一个字符向量。字符向量的类是 `character`：

```
class(c("she", "sells", "seashells", "on", "the", "sea", "shore"))
```

```
## [1] "character"
```

请注意，和某些语言不同，R 不区分整个字符串和单个字符——只包含一个字符的字符串与其他字符串的处理相同。与一些低级语言不同，你无需用空字符（`\0`）来终止字符串。事实上，把这样一个字符加进字符串中也是错误的。

在许多编程语言中，类别数据用整数表示。例如，gender 中用 `1` 来代表 female，`2` 代表 male。稍好的办法是把 gender 当作带有“female”和“male”选项的字符变量。然而，因为类别数据与传统的纯文本是不同的概念，所以从语义上看这仍然不妥。R 找到了一种更有效的方法，把这两种方法整合到一个语义正确的类里面——因子（factor），即拥有标签的整数：

```
(gender <- factor(c("male", "female", "female", "male", "female")))
```

```
## [1] male female female male female
## Levels: female male
```

因子的内容看起来与它们所对应的字符一样——这样每个值都能得到一个可读性很好的标签。这些标签被限制在称为因子水平（levels of the factor）的特定值中（本例中为“female”和“male”）：

```
levels(gender)
```

```
## [1] "female" "male"
```

```
nlevels(gender)
```

```
## [1] 2
```

请注意，即使“male”是 gender 中的第一个值，第一个水平仍是“female”。默认情况下，因子水平按字母顺序分配。

在底层，因子的值被存储为整数而非字符。你可通过调用 as.integer 清楚地看到：

```
as.integer(gender)

## [1] 2 1 1 2 1
```

采取整数而非字符文本的存储方式，令内存的使用非常高效，尤其当出现大量重复字符串时。如果我们再夸张一点，生成 10 000 个随机的 gender 值（使用 sample 函数对字符串“female”和“male”随机采样 10 000 次并使用 replace 选项），可以看到，一个因子包含的值比同等字符占用更少的内存。在以下代码中，sample 返回一个字符向量（这是使用 as.factor 转换而成的），而 object.size 则返回每个对象的内存分配大小：

```
gender_char <- sample(c("female", "male"), 10000, replace = TRUE)
gender_fac <- as.factor(gender_char)
object.size(gender_char)

## 80136 bytes

object.size(gender_fac)

## 40512 bytes
```

32 位和 64 位系统中变量所占用的内存数量是不一样的，所以在不同情况下 object.size 将返回不同的值。

当操作因子水平的内容时（常见的例子是：清理命名，把所有的男性字符统一为“male”而非“Male”），最好先把因子转换成字符串后再处理，以便充分利用字符串操作函数。可以使用 as.character 函数完成转换：

```
as.character(gender)

## [1] "male" "female" "female" "male" "female"
```

更多有关字符向量和因子的内容将在第 7 章中深入探讨。

原始类 raw 存储向量的“原始”字节[2]。每个字节由一个两位的十六进制值表示。它们主要用于保存输入的二进制文件的内容，因而比较少见。使用 as.raw 函数可把 0 到 255 之间的整数转换为原始值（raw）。此范围之外的数字将全部视为 0，分数和虚部也被丢弃。对于字符串，as.raw 则不起作用，而须使用 charToRaw 函数：

注 2：不确定是什么字节。

```
as.raw(1:17)

## [1] 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10 11

as.raw(c(pi, 1 + 1i, -1, 256))

## Warning: imaginary parts discarded in coercion

## Warning: out-of-range values treated as 0 in coercion to raw

## [1] 03 01 00 00

(sushi <- charToRaw("Fish!"))

## [1] 46 69 73 68 21

class(sushi)

## [1] "raw"
```

除了目前我们已了解到的向量类，还有许多其他类型的变量。我们将在接下来的章节中继续讨论它们。

数组包含多维数据，矩阵（通过 `matrix` 类）是特殊的二维数组，这些将在第 4 章中讨论。

目前，所有这些变量类型都要包含相同类型的值。例如，字符（`character`）向量或数组里须包含字符串，逻辑（`logical`）向量或数组须只能包含逻辑值。但列表（list）则不一样，它比较灵活，列表里的每一项都可以是不同的类型，甚至能包含其他列表。数据框（data frame）像是矩阵和列表的共同产物。它既像矩阵一样是矩形的，又像列表一样，每一列都可以有不同的类型。它们非常适合于存储类似电子表格这样的数据。第 5 章将继续讨论列表和数据框。

前面的类都用于存储数据。环境（environment）中会存储那些能保存数据的变量。很显然，除了保存数据之外，我们也需要函数来和它一起工作，如之前提过的函数 `sin` 和 `exp`。事实上，像 + 这样的操作符也是隐藏的函数！第 6 章将进一步谈及环境和函数。

第 7 章将深入讨论字符串和因子，以及用于存储日期和时间的一些选项。

R 中还有一些稍难理解的类型，留待后文讨论。第 15 章将讨论公式（formuale）；16.5 节将讨论调用（call）和表达式（expression）；16.6 节则将进一步探讨类。

3.5　检查和更改类

直接在命令提示符下，以交互的方式键入类 `class` 的函数名来检查变量是很有用的。但是如果想在脚本中测试对象的类型，则最好用 `is` 函数，或针对某个类写的特定函数。通常，我们会做这样的测试：

```
if(!is(x, "some_class"))
{
  # 采取某些纠正措施
}
```

大部分的类都会有自己的 is.* 函数。通常，直接调用它们比使用通用 is 函数稍显高效。例如：

```
is.character("red lorry, yellow lorry")
```

```
## [1] TRUE
```

```
is.logical(FALSE)
```

```
## [1] TRUE
```

```
is.list(list(a = 1, b = 2))
```

```
## [1] TRUE
```

以下命令可查看在 base 包中所有的 is 函数：

```
ls(pattern = "^is", baseenv())
```

```
##  [1] "is.array"              "is.atomic"
##  [3] "is.call"               "is.character"
##  [5] "is.complex"            "is.data.frame"
##  [7] "is.double"             "is.element"
##  [9] "is.environment"        "is.expression"
## [11] "is.factor"             "is.finite"
## [13] "is.function"           "is.infinite"
## [15] "is.integer"            "is.language"
## [17] "is.list"               "is.loaded"
## [19] "is.logical"            "is.matrix"
## [21] "is.na"                 "is.na.data.frame"
## [23] "is.na.numeric_version" "is.na.POSIXlt"
## [25] "is.na<-"               "is.na<-.default"
## [27] "is.na<-.factor"        "is.name"
## [29] "is.nan"                "is.null"
## [31] "is.numeric"            "is.numeric.Date"
## [33] "is.numeric.difftime"   "is.numeric.POSIXt"
## [35] "is.numeric_version"    "is.object"
## [37] "is.ordered"            "is.package_version"
## [39] "is.pairlist"           "is.primitive"
## [41] "is.qr"                 "is.R"
## [43] "is.raw"                "is.recursive"
## [45] "is.single"             "is.symbol"
## [47] "is.table"              "is.unsorted"
## [49] "is.vector"             "isatty"
## [51] "isBaseNamespace"       "isdebugged"
## [53] "isIncomplete"          "isNamespace"
## [55] "isOpen"                "isRestart"
## [57] "isS4"                  "isSeekable"
## [59] "isSymmetric"           "isSymmetric.matrix"
## [61] "isTRUE"
```

在上例中，`ls` 列出所有的变量名，`"^is"` 是一个正则表达式，它意味着“匹配所有以 `'is'` 开头的字符串”，而 `baseenv` 函数则返回 `base` 包中所有的环境。环境是相对高级的话题，将在第 6 章讨论，现在不用拘泥于它的含义。

`assertive` 包[3]含有更多 `is` 函数，命名方式更加一致。

有点奇怪的是，`is.numeric` 函数对整数和浮点数都返回 `TRUE` 值。如果我们只测试浮点数，则须使用 `is.double`。然而，一般无需这么做，因为在 R 中浮点和整数几乎能互换使用。请注意，在下例中，在数字后面添加 `L` 后缀就能把它转换为整数：

```
is.numeric(1)

## [1] TRUE

is.numeric(1L)

## [1] TRUE

is.integer(1)

## [1] FALSE

is.integer(1L)

## [1] TRUE

is.double(1)

## [1] TRUE

is.double(1L)

## [1] FALSE
```

有时候，我们想改变一个对象的类型。这就是所谓的*转型*（casting），大部分的 `is*` 函数都有与之对应的 `as*` 函数。尽可能使用特定的 `as*` 函数而非单纯的 `as` 函数，因为它们通常更有效，而且往往针对该类会有额外的逻辑。例如，当尝试把字符串转换为数字时，`as.numeric` 比单独的 `as` 函数效率稍高，虽然两者皆可使用：

```
x <- "123.456"
as(x, "numeric")

## [1] 123.5

as.numeric(x)

## [1] 123.5
```

注 3：这个包是我写的。

R 能打印到小数点后第几位取决于 R 的设置。你可以使用 `options(digits=n)` 来设置全局默认选项，其中 n 在 1 和 22 之间。第 7 章将再讨论如何控制打印数字。

然而，请注意在下例中，当一个向量被转换成一个数据框时（类似电子表格的数据变量），`as` 函数抛出一个错误：

```
y <- c(2, 12, 343, 34997)       # 请参考 http://oeis.org/A192892
as(y, "data.frame")
as.data.frame(y)
```

一般情况下，特定类的变量的使用应始终优先于标准函数 `as`。

尽管不推荐（见 16.7 节，类的分配另有用途），我们也可以直接给对象分配一个新的类以改变其类型：

```
x <- "123.456"
class(x) <- "numeric"
x

## [1] 123.5

is.numeric(x)

## [1] TRUE
```

3.6 检查变量

当在控制台输入一个运算或者变量时，结果就被打印出来，因为 R 隐式调用了对象的 `print` 方法。

对以下术语稍作解释：大多情况下，“方法”（method）和“函数”（function）是可以互换的。有时，R 中的函数也称为面向对象中的方法。不同类型的对象有不同版本的 `print` 函数，这使得矩阵与向量的打印方法不同，这就是谈到“打印方法”的原因。

所以，在命令提示符下输入 `1 + 1` 与 `print( 1 + 1)` 一样。

但是，对于内循环或函数来说[4]，自动打印功能不起作用，我们必须显式地调用 `print`：

注 4：除非函数返回了值。

```
ulams_spiral <- c(1, 8, 23, 46, 77)  # 参考 http://oeis.org/A033951
for(i in ulams_spiral) i             # 啊哦，值没有打印出来
for(i in ulams_spiral) print(i)

## [1] 1
## [1] 8
## [1] 23
## [1] 46
## [1] 77
```

如果你是在终端而不是在 GUI 或 IDE 上运行 R，在某些系统上这个问题的确存在。在这种情况下，你就一直需要显式调用 `print` 函数。

大部分 `print` 函数的实现建立在调用底层的 `cat` 函数上。虽然你可能永远无需直接调用 `cat`（与之对应的用户级命令是 `print` 和 `message`），但是仍应对此有所了解，以便在需要时亲自编写 `print` 函数[5]。

> `c` 和 `cat` 函数都是 concatenate 的缩写，但它们的作用完全不同！ `cat` 是以 Unix 中的函数命名的。

除了查看变量的打印输出，最好也能看到某种程度的对象汇总信息。`summary` 函数就能为不同的数据类型提供汇总信息。例如，数值变量会被汇总统计出平均数、中位数，以及一些分位数（quantile）。在下例中，`runif` 函将数生成 30 个均匀分布于 0 和 1 之间的随机数：

```
num <- runif(30)
summary(num)

#    Min.  1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.0211  0.2960  0.5060  0.5290  0.7810  0.9920
```

类别变量和逻辑向量将根据每个值的计算进行汇总。在下例中，`letters` 是一个内置的常数，它包含了从 a 到 z 的小写值（大写的 `LETTERS` 则包括了类似的从 A 到 Z 的大写值）。这里 `letters[ 1:5]` 用索引限制 `letters` 的范围为从 a 到 e。`sample` 函数使用重复抽样 `replace` 随机抽样 `30` 次：

```
fac <- factor(sample(letters[1:5], 30, replace = TRUE))
summary(fac)

## a b c d e
## 6 7 5 9 3

bool <- sample(c(TRUE, FALSE, NA), 30, replace = TRUE)
summary(bool)

##    Mode   FALSE    TRUE    NA's
## logical      12      11       7
```

注 5：也许就像在练习 16-3 中那样。

多维对象与矩阵及数据框一样，都是通过列来汇总的（第 4 章和第 5 章将做更详细的讨论）。我们在下面创建的数据框 dfr 非常大，显示出来足足有 30 行。对于这样的庞然大物[6]，用 head 函数即可仅显示它的前几行（默认为 6 行）：

```
dfr <- data.frame(num, fac, bool)
head(dfr)

##       num fac  bool
## 1 0.47316   b    NA
## 2 0.56782   d FALSE
## 3 0.46205   d FALSE
## 4 0.02114   b  TRUE
## 5 0.27963   a  TRUE
## 6 0.46690   a  TRUE
```

数据框的 summary 函数就像为每列单独调用 summary 一样：

```
summary(dfr)

##      num          fac       bool
## Min.   :0.0211    a:6   Mode :logical
## 1st Qu.:0.2958    b:7   FALSE:12
## Median :0.5061    c:5   TRUE :11
## Mean   :0.5285    d:9   NA's :7
## 3rd Qu.:0.7808    e:3
## Max.   :0.9916
```

类似地，str 函数能显示对象的结构。对向量来说，它并非很有趣（因为它们太简单了），但 str 对数据框和嵌套列表非常有用：

```
str(num)
## num [1:30] 0.4732 0.5678 0.462 0.0211 0.2796 ...

str(dfr)
## 'data.frame':        30 obs. of  3 variables:
##  $ num : num  0.4732 0.5678 0.462 0.0211 0.2796 ...
##  $ fac : Factor w/ 5 levels "a","b","c","d",..: 2 4 4 2 1 1 4 2 1 4 ...
##  $ bool: logi  NA FALSE FALSE TRUE TRUE TRUE ...
```

如前所述，每个类都有自己的打印（print）方法，以此控制如何显示到控制台。有时，这种打印模糊了其内部结构，或忽略了一些有用的信息。用 unclass 函数可绕开这一点，显示变量是如何构建的。例如，对因子调用 unclass 函数会显示它仅是一个整数（integer）向量，拥有一个叫 levels 的属性：

```
unclass(fac)

## [1] 2 4 4 2 1 1 4 2 1 4 3 3 1 5 4 5 1 5 1 2 2 3 4 2 4 3 4 2 3 4
## attr(,"levels")
## [1] "a" "b" "c" "d" "e"
```

注 6：现在，30 行的数据算不上是“大数据”，但在打印时它仍能占满整个屏幕。

稍后我们会讨论属性，而现在须了解的是，attributes 函数能显示当前对像的所有属性列表：

```
attributes(fac)

## $levels
## [1] "a" "b" "c" "d" "e"
##
## $class
## [1] "factor"
```

为了可视化诸如矩阵和数据框之类的二维变量，View（请注意大写字母“V”）函数会把变量（只读的）显示为电子表格（spreadsheet）。edit 和 fix 函数的工作方式与 View 类似，但它允许手动更改数据值。虽然这听似更实用，但以这种方式编辑数据却是个无比糟糕的主意，因为我们会失去所有的可追溯性而无法追踪数据的出处。最好的方式还是使用编程来编辑数据：

```
View(dfr)                     # 不允许更改
new_dfr <- edit(dfr)          # 更改将保存于 new_dfr
fix(dfr)                      # 更改将保存于 dfr
```

一个好方法是结合 View 和 head 函数来查看数据框的前几行：

```
View(head(dfr, 50)) # 查看前 50 行
```

3.7 工作区

工作时，我们往往想知道已经创建的变量及其内容。用 ls 函数即可列出现有变量的名称。它是以与其类似的 Unix 命令命名的，并且遵循相同的约定：默认情况下，变量名以 . 开头的是隐藏文件。要查看它们，可传入 all.names=TRUE 参数：

```
# 创建一些变量以便查找
peach <- 1
plum <- "fruity"
pear <- TRUE
ls()

##  [1] "a_vector"          "all_true"          "bool"
##  [4] "dfr"               "fac"               "fname"
##  [7] "gender"            "gender_char"       "gender_fac"
## [10] "i"                 "input"             "my_local_variable"
## [13] "none_true"         "num"               "output"
## [16] "peach"             "pear"              "plum"
## [19] "remove_package"    "some_true"         "sushi"
## [22] "ulams_spiral"      "x"                 "xy"
## [25] "y"                 "z"                 "zz"

ls(pattern = "ea")

## [1] "peach" "pear"
```

要了解更多工作区中的信息，可使用 `ls.str` 函数查看变量的结构。可能正如你所料，它是 `ls` 和 `str` 函数的结合，且它在调试会话（session）中作用很大（见 16.4 节）。`browseEnv` 提供类似的功能，但它在网页浏览器中以 HTML 页面的格式显示其输出：

```
browseEnv()
```

工作一段时间后，尤其在数据挖掘中，工作区会变得相当凌乱，我们可以使用 `rm` 函数删除变量来清理区间：

```
rm(peach, plum, pear)
rm(list = ls())        # 删除所有变量。小心使用！
```

3.8 小结

- 所有的变量都有一个类。
- 可通过 `is` 函数或特定类的变量来测试一个对象是否是某个类。
- 可使用 `as` 函数或特定类的变量来改变一个对象的类。
- 有几个函数可用于检视变量，其中包括 `summary`、`head`、`str`、`unclass`、`attributes` 和 `View`。
- `ls` 能列出你的变量；`ls.str` 则连同名字及结构一起列出。
- `rm` 能删除变量。

3.9 知识测试：问题

- 问题 3-1
  数字的三个内置类的名称是什么？

- 问题 3-2
  用什么函数查找了因子的水平值？

- 问题 3-3
  如何把字符串“6.283185”转换为数字？

- 问题 3-4
  指出至少三个用于检视变量内容的函数。

- 问题 3-5
  如何删除用户工作区中的所有变量？

3.10 知识测试：练习

- 练习 3-1
  查找以下值 `Inf`、`NA`、`NaN` 和 `""` 的类、类型、模式及存储模式。[5]

- 练习 3-2
  随机从“dog”、“cat”、“hamster”和“goldfish”中以相等的概率生成 1000 个宠物名。显示所得变量的前几个值，并计算每种宠物的数量。[5]

- 练习 3-3
  创建一些以蔬菜命名的变量。列出用户工作区中所有包含字母“a”的变量。[5]

第 4 章

向量、矩阵和数组

第 1 章和第 2 章介绍了几种向量类型：逻辑值向量、字符串向量和数字向量。本章将介绍更多向量的操作方法，以及它们的多维兄弟：矩阵和数组。

4.1　本章目标

阅读完本章后，你会了解以下内容：

- 如何从现有向量中创建出新的向量；
- 长度、维度和命名；
- 如何创建、操纵矩阵和数组。

4.2　向量

现在，你已经试过用冒号运算符 : 来创建从某个数到另一个数的数字序列，以及用 c 函数来拼接数值和向量，以创建更长的向量。总结如下：

```
8.5:4.5                   # 从 8.5 到 4.5 的数字序列

## [1] 8.5 7.5 6.5 5.5 4.5

c(1, 1:3, c(5, 8), 13) # 不同值被拼接成单一向量

## [1]  1  1  2  3  5  8 13
```

vector 函数能创建一个指定类型和长度的矢量。其结果中的值可为零、FALSE、空字符串，

或任何相当于“什么都没有”（nothing）的类型：

```
vector("numeric", 5)
```

```
## [1] 0 0 0 0 0
```

```
vector("complex", 5)
```

```
## [1] 0+0i 0+0i 0+0i 0+0i 0+0i
```

```
vector("logical", 5)
```

```
## [1] FALSE FALSE FALSE FALSE FALSE
```

```
vector("character", 5)
```

```
## [1] "" "" "" "" ""
```

```
vector("list", 5)
```

```
## [[1]]
## NULL
##
## [[2]]
## NULL
##
## [[3]]
## NULL
##
## [[4]]
## NULL
##
## [[5]]
## NULL
```

在上例中，NULL 是一个特殊的“空”值（不要与代表缺失值的 NA 混淆）。第 5 章会详细讨论 NULL。为方便起见，用每个类型的包装（wrapper）函数来创建矢量以节省打字时间。下列命令与上面代码中的命令是等价的：

```
numeric(5)
```

```
## [1] 0 0 0 0 0
```

```
complex(5)
```

```
## [1] 0+0i 0+0i 0+0i 0+0i 0+0i
```

```
logical(5)
```

```
## [1] FALSE FALSE FALSE FALSE FALSE
```

```
character(5)
```

```
## [1] "" "" "" "" ""
```

在下一章中我们将看到，list 函数的工作方式与它们不一样。list(5) 创建的内容有所不同。

4.2.1 序列

除了冒号运算符之外，还有几个其他函数能创建更为通用的序列。其中，seq 函数最常见，能以许多不同的方式指定序列。然而在实际中，你不需要调用它，因为有三个专门的序列函数，运行更快且更易用，满足了特定场合的使用。

seq.int 可创建一个序列，序列的范围由两个数字指定。只需两个参数，原理与冒号运算符完全相同：

```
seq.int(3, 12)     # 与 3:12 相同

##  [1]  3  4  5  6  7  8  9 10 11 12
```

seq.int 稍微比 : 更通用些，因为它可以指定步长：

```
seq.int(3, 12, 2)

## [1]  3  5  7  9 11

seq.int(0.1, 0.01, -0.01)

##  [1] 0.10 0.09 0.08 0.07 0.06 0.05 0.04 0.03 0.02 0.01
```

因为 seq_len 函数将创建一个从 1 到它的输入值的序列，所以用 seq_len(5) 来表达 1:5 有些笨拙。不过，当输入值为零时，该函数非常有用：

```
n <- 0

1:n          # 和你预期的可能不一样!

## [1] 1 0

seq_len(n)

## integer(0)
```

seq_along 创建一个从 1 开始、长度为其输入值的序列：

```
pp <- c("Peter", "Piper", "picked", "a", "peck", "of", "pickled", "peppers")
for(i in seq_along(pp)) print(pp[i])

## [1] "Peter"
## [1] "Piper"
## [1] "picked"
## [1] "a"
```

```
## [1] "peck"
## [1] "of"
## [1] "pickled"
## [1] "peppers"
```

在以上的每一个例子中，你都可以简单地用 seq 替换 seq.int、seq_len 或 seq_along，得到的结果相同，虽然这样做并无必要。

4.2.2 长度

我们刚才已经悄悄地引入了一个关于向量的新概念：所有的向量都有一个长度，它告诉我们向量包含多少个元素。这是一个非负整数[1]（是的，向量的长度可以为零），你可以通过 length 函数查到这个值。缺失值也会被计入长度：

```
length(1:5)
```

```
## [1] 5
```

```
length(c(TRUE, FALSE, NA))
```

```
## [1] 3
```

容易造成混淆的地方是字符向量。它们的长度为字符串的数目，而非每个字符串中字符数的长度。对此，我们使用 nchar：

```
sn <- c("Sheena", "leads", "Sheila", "needs")
length(sn)
```

```
## [1] 4
```

```
nchar(sn)
```

```
## [1] 6 5 6 5
```

我们也可以为向量重新分配一个长度，不过很少这么做，且几近意味着糟糕的代码。例如，如果向量的长度缩短，那么后面的值将会被删除；而如果长度增加，则缺失值会添加到最后：

```
poincare <- c(1, 0, 0, 0, 2, 0, 2, 0) # 请参见：http://oeis.org/A051629
length(poincare) <- 3
poincare
```

```
## [1] 1 0 0
```

```
length(poincare) <- 8
poincare
```

```
## [1]  1  0  0 NA NA NA NA NA
```

注 1：在 32 位的系统以及 3.0.0 之前版本的 R 中，长度被限制为 2^31-1 个元素。

4.2.3 命名

R 向量的一大特性是能给每个元素命名。通常，标记元素可使代码可读性更强。你可以使用 `name = value` 的形式在创建向量时为其指定名称。如果元素的名称是有效的，则无需被引号括起来。你也可以只命名向量中的某些元素而忽略其他元素：

```
c(apple = 1, banana = 2, "kiwi fruit" = 3, 4)

##      apple     banana kiwi fruit
##          1          2          3          4
```

你也可以在向量创建后用 `names` 函数为元素添加名字：

```
x <- 1:4
names(x) <- c("apple", "bananas", "kiwi fruit", "")
x

##      apple    bananas kiwi fruit
##          1          2          3          4
```

此 `name` 函数也可用于取得向量的名称：

```
names(x)

## [1] "apple"      "bananas"    "kiwi fruit" ""
```

如果向量中没有一个元素有名字，则 `names` 函数返回 `NULL`：

```
names(1:4)

## NULL
```

4.2.4 索引向量

通常，我们只要访问向量中的部分或个别元素。这就是所谓的*索引*，它用方括号 `[]` 来实现。（有人也称之为*子集*、*下标*或*切片*，这些术语所指相同。）R 系统非常灵活，提供如下多种索引方法。

- 给向量传入正数，它会返回此位置上的向量元素切片。它的第一个位置是 1（而不像其他某些语言一样是 0）。
- 给向量传入负数，它会返回一个向量切片，它将包含除了这些位置以外的所有元素。
- 给向量传入一个逻辑向量，它会返回一个向量切片，里面只包含索引为 `TRUE` 的元素。
- 对于已命名的向量，给向量传入命名的字符向量，将会返回向量中包含这些名字的元素切片。

请看以下向量：

```
x <- (1:5) ^ 2

## [1]  1  4  9 16 25
```

以下三个索引方法都将返回相同的值：

```
x[c(1, 3, 5)]

x[c(-2, -4)]

x[c(TRUE, FALSE, TRUE, FALSE, TRUE)]

## [1]  1  9 25
```

如果给每个元素命名，以下方法也将返回相同的值：

```
names(x) <- c("one", "four", "nine", "sixteen", "twenty five")
x[c("one", "nine", "twenty five")]

##         one        nine twenty five
##           1           9          25
```

混合使用正负值是不允许的，会抛出一个错误：

```
x[c(1, -1)]     # 这说不通！

## Error: only 0's may be mixed with negative subscripts
```

如果你使用正数或逻辑值作为下标，那么缺失索引所对应的值同样也是缺失值：

```
x[c(1, NA, 5)]
##         one        <NA> twenty five
##           1          NA          25

x[c(TRUE, FALSE, NA, FALSE, TRUE)]
##         one        <NA> twenty five
##           1          NA          25
```

对于负的下标值来说，不允许存在缺失值，会产生错误：

```
x[c(-2, NA)]     # 这也讲不通！

## Error: only 0's may be mixed with negative subscripts
```

超出范围的下标值（即超出了矢量的长度）不会导致错误，而是返回缺失值 `NA`。在实际中，最好确保你的下标值都在使用范围内：

```
x[6]

## <NA>
##   NA
```

非整数下标会默认向零舍入。这是另一个 R 被认为过于宽松的证例。如果你传入的下标是分数，那么很可能你正在写一些糟糕的代码：

```
x[1.9]   #1.9 舍入为 1
## one
## 1
```

```
x[-1.9]  #-1.9 舍入为 -1
```

```
##        four        nine     sixteen twenty five
##           4           9          16          25
```

不传递任何下标值将返回整个向量。不过再次提醒，如果你没有传递任何索引值，那么很可能你正在做一些不靠谱的事情：

```
x[]
```

```
##         one        four        nine     sixteen twenty five
##           1           4           9          16          25
```

`which` 函数将返回逻辑向量中为 `TRUE` 的位置。如果要将逻辑索引切换到整数索引中，这个函数很有用：

```
which(x > 10)
```

```
##     sixteen twenty five
##           4           5
```

`which.min` 和 `which.max` 分别是 `which(min(x))` 和 `which(max(x))` 的简写：

```
which.min(x)
```

```
## one
##   1
```

```
which.max(x)
```

```
## twenty five
##           5
```

4.2.5　向量循环和重复

到目前为止，所有相加在一起的向量都具有相同的长度。你可能会问：“当我对不同长度的向量做运算，结果会如何？”

如果我们把一个单独的数字与向量相加，则向量的每个元素都会与该数字相加：

```
1:5 + 1
```

```
## [1] 2 3 4 5 6
```

```
1 + 1:5
```

```
## [1] 2 3 4 5 6
```

将两个向量相加时，R 将会循环较短向量中的元素以配合较长的那个：

```
1:5 + 1:15
```

```
##  [1]  2  4  6  8 10  7  9 11 13 15 12 14 16 18 20
```

如果长向量的长度不是短向量长度的倍数，将出现一个警告：

```
1:5 + 1:7
```

```
## Warning: longer object length is not a multiple of shorter object length
```

```
## [1]  2  4  6  8 10  7  9
```

必须强调的是，虽然我们可以在不同长度的向量之间做运算，但这并不意味着*应该*这样做。为向量添加一个标量值没有问题，但我们会在其他方面把自己搞晕。更好的做法是明确地创建同等长度的向量，然后再对它们进行操作。

`rep` 函数非常适合此类任务，它允许我们重复使用元素来创建矢量：

```
rep(1:5, 3)
```

```
##  [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
```

```
rep(1:5, each = 3)
```

```
##  [1] 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
```

```
rep(1:5, times = 1:5)
```

```
##  [1] 1 2 2 3 3 3 4 4 4 4 5 5 5 5 5
```

```
rep(1:5, length.out = 7)
```

```
## [1] 1 2 3 4 5 1 2
```

正如 `seq` 函数一样，`rep` 有一个更为简单和快速的变体 `rep.int`：

```
rep.int(1:5, 3)  # 与 rep(1:5, 3) 一样
```

```
##  [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
```

类似 `seq_len`，在最近的 R 版本（从 v3.0.0 开始）中也有 `rep_len` 函数，可指定输出向量的长度：

```
rep_len(1:5, 13)
```

```
##  [1] 1 2 3 4 5 1 2 3 4 5 1 2 3
```

4.3 矩阵和数组

到目前为止，我们见到的变量都是一维的，这是因为它们只有长度而没有其他维度。数组能存放多维矩形数据。“矩形”是指每行的长度都相等，且每列和其他长度也是如此。矩阵是二维数组的特例。

4.3.1 创建数组和矩阵

可以使用 `array` 函数创建一个数组，为它们传入两个向量（值和维度）作为参数。另外，你也可以为每个维度命名：

```
(three_d_array <- array(
  1:24,
  dim = c(4, 3, 2),
  dimnames = list(
    c("one", "two", "three", "four"),
    c("ein", "zwei", "drei"),
    c("un", "deux")
  )
))

## , , un
##
##       ein zwei drei
## one     1    5    9
## two     2    6   10
## three   3    7   11
## four    4    8   12
##
## , , deux
##
##       ein zwei drei
## one    13   17   21
## two    14   18   22
## three  15   19   23
## four   16   20   24

class(three_d_array)

## [1] "array"
```

创建矩阵的语法非常类似，但无需传递维度 `dim` 参数，只要指定行数或列数即可：

```
(a_matrix <- matrix(
  1:12,
  nrow = 4,     #ncol = 3 也是同样的效果
  dimnames = list(
    c("one", "two", "three", "four"),
    c("ein", "zwei", "drei")
  )
))
```

```
##      ein zwei drei
##one     1    5    9
##two     2    6   10
##three   3    7   11
##four    4    8   12
```

```
class(a_matrix)
```

```
## [1] "matrix"
```

该矩阵也可以用 array 函数来创建。下列二维数组与刚创建的矩阵（它甚至有矩阵类 matrix）完全相等：

```
(two_d_array <- array(
  1:12,
  dim = c(4, 3),
  dimnames = list(
    c("one", "two", "three", "four"),
    c("ein", "zwei", "drei")
  )
))
```

```
##      ein zwei drei
##one     1    5    9
##two     2    6   10
##three   3    7   11
##four    4    8   12
```

```
identical(two_d_array, a_matrix)
```

```
## [1] TRUE
```

```
class(two_d_array)
```

```
## [1] "matrix"
```

创建矩阵时，传入的值会按列填充矩阵。也可指定参数 byrow = TRUE 来按行填充矩阵：

```
matrix(
  1:12,
  nrow = 4,
  byrow = TRUE,
  dimnames = list(
    c("one", "two", "three", "four"),
    c("ein", "zwei", "drei")
  )
)
```

```
##      ein zwei drei
##one     1    2    3
##two     4    5    6
##three   7    8    9
##four   10   11   12
```

4.3.2 行、列和维度

对于矩阵和数组，dim 函数将返回其维度的整数值向量：

```
dim(three_d_array)

## [1] 4 3 2

dim(a_matrix)

## [1] 4 3
```

对于矩阵，函数 nrow 和 ncol 将分别返回行数和列数：

```
nrow(a_matrix)

## [1] 4

ncol(a_matrix)

## [1] 3
```

nrow 和 ncol 也能用于数组，分别返回第一和第二个维度。但对于更高维度的对象，通常最好使用 dim 函数：

```
nrow(three_d_array)

## [1] 4

ncol(three_d_array)

## [1] 3
```

之前使用过的 length 函数也能用于矩阵和数组。在这种情况下，它将返回所有维度的乘积：

```
length(three_d_array)

## [1] 24

length(a_matrix)

## [1] 12
```

还可通过给 dim 函数分配一个新的维度来重塑矩阵或数组。不过请小心使用，因为它会删除原维度的名称：

```
dim(a_matrix) <- c(6, 2)
a_matrix

##      [,1] [,2]
##[1,]    1    7
##[2,]    2    8
```

```
##[3,]    3    9
##[4,]    4   10
##[5,]    5   11
##[6,]    6   12
```

把 nrow、ncol 和 dim 用于向量时将返回 NULL 值。与 nrow 和 ncol 相对的函数是 NROW 和 NCOL，它们把向量看做具有单个列的矩阵（也即数学意义上的列向量）：

```
identical(nrow(a_matrix), NROW(a_matrix))

## [1] TRUE

identical(ncol(a_matrix), NCOL(a_matrix))

## [1] TRUE

recaman <- c(0, 1, 3, 6, 2, 7, 13, 20)
nrow(recaman)
## NULL
NROW(recaman)
## [1] 8
ncol(recaman)
## NULL
NCOL(recaman)
## [1] 1
dim(recaman)
```

4.3.3　行名、列名和维度名

就像向量中的元素都有名称 names 一样，矩阵的行和列也有行名 rownames 和列名 colnames。出于历史的原因，还有一个 row.names 函数，它的作用与 rownames 函数相同。不过因为没有相应的 col.names，所以最好还是忽略 row.names，而只使用 rownames。至于 nrow、ncol 和 dim 函数，它们对应于数组的函数是 dimnames。后者将返回一个字符向量列表（参见 5.2 节）。在以下的代码中，a_matrix 已恢复到其维度被改变之前的状态：

```
rownames(a_matrix)

## [1] "one"   "two"   "three" "four"

colnames(a_matrix)

## [1] "ein"  "zwei" "drei"

dimnames(a_matrix)

## [[1]]
## [1] "one"   "two"   "three" "four"
##
## [[2]]
## [1] "ein"  "zwei" "drei"
```

```
rownames(three_d_array)

## [1] "one"   "two"   "three" "four"

colnames(three_d_array)

## [1] "ein"  "zwei" "drei"

dimnames(three_d_array)

## [[1]]
## [1] "one"   "two"   "three" "four"
##
## [[2]]
## [1] "ein"  "zwei" "drei"
##
## [[3]]
## [1] "un"   "deux"
```

4.3.4　索引数组

索引数组与索引向量类似，只是现在要索引的维度多于一个。和之前一样，我们用方括号来表示索引，且仍有四种指定索引的方法（正整数、负整数、逻辑值和元素的名称）。在不同的维度上用不同的方式指定索引下标完全没问题。每个维度的下标用逗号分隔：

```
a_matrix[1, c("zwei", "drei")]     # 在第一行、第二列和第三列的两个元素

## zwei drei
##    5    9
```

要包括所有维度，则只需置空相应的下标：

```
a_matrix[1, ]                     # 第一行的所有元素

## ein zwei drei
##   1    5    9

a_matrix[, c("zwei", "drei")]     # 第二列、第三列的所有元素

##       zwei drei
## one      5    9
## two      6   10
## three    7   11
## four     8   12
```

4.3.5　合并矩阵

c 函数能在拼接矩阵之前把它们转换成向量：

```
(another_matrix <- matrix(
  seq.int(2, 24, 2),
  nrow = 4,
```

```
  dimnames = list(
    c("five", "six", "seven", "eight"),
    c("vier", "funf", "sechs")
  )
))

##       vier funf sechs
## five     2   10    18
## six      4   12    20
## seven    6   14    22
## eight    8   16    24

c(a_matrix, another_matrix)

##  [1]  1  2  3  4  5  6  7  8  9 10 11 12  2  4  6  8 10 12 14 16 18 20 22
## [24] 24
```

通过使用 `cbind` 和 `rbind` 函数按行和列来绑定两个矩阵，能更自然地合并它们：

```
cbind(a_matrix, another_matrix)

##       ein zwei drei vier funf sechs
## one     1    5    9    2   10    18
## two     2    6   10    4   12    20
## three   3    7   11    6   14    22
## four    4    8   12    8   16    24

rbind(a_matrix, another_matrix)

##       ein zwei drei
## one     1    5    9
## two     2    6   10
## three   3    7   11
## four    4    8   12
## five    2   10   18
## six     4   12   20
## seven   6   14   22
## eight   8   16   24
```

4.3.6 数组算术

和向量中的运算一样，标准算术运算符（`+`、`-`、`\*`、`/`）将以同样的方式按元素来处理矩阵和数组：

```
a_matrix + another_matrix

##       ein zwei drei
## one     3   15   27
## two     6   18   30
## three   9   21   33
## four   12   24   36
```

```
a_matrix * another_matrix

##       ein zwei drei
## one     2   50  162
## two     8   72  200
## three  18   98  242
## four   32  128  288
```

当对两个数组执行算术运算时，须确保它们的大小适当（以线性代数的术语来说，它们必须是“一致的”）。例如，两个数组在相加时大小必须相等，而矩阵相乘时第一个矩阵的行数必须和第二矩阵的列数相等：

```
(another_matrix <- matrix(1:12, nrow = 2))
a_matrix + another_matrix         #将两个不一致的矩阵相加会抛出错误
```

如果你尝试把向量和数组相加，那么通常的向量循环规则是适用的，但结果的维度只取自数组。

t 函数用于转置矩阵（但不能转置更高维的数组，其中的概念没有被明确定义）：

```
t(a_matrix)
##      one two three four
## ein    1   2     3    4
## zwei   5   6     7    8
## drei   9  10    11   12
```

对于矩阵内乘和外乘运算，我们有特殊运算符 %*% 和 %o%。每一次，如果维度的名称存在的话，都取自第一个输入：

```
a_matrix %*% t(a_matrix)     #内乘

##       one two three four
## one   107 122   137  152
## two   122 140   158  176
## three 137 158   179  200
## four  152 176   200  224

1:3 %o% 4:6                  #外乘

##      [,1] [,2] [,3]
## [1,]    4    5    6
## [2,]    8   10   12
## [3,]   12   15   18

outer(1:3, 4:6)              #结果一样

##      [,1] [,2] [,3]
## [1,]    4    5    6
## [2,]    8   10   12
## [3,]   12   15   18
```

因为幂运算符 ^ 也将作用于矩阵中的每个元素，所以在为矩阵求反时不能简单地为矩阵使用负一次方运算，而应使用 solve 函数[2]：

```
(m <- matrix(c(1, 0, 1, 5, -3, 1, 2, 4, 7), nrow = 3))

##      [,1] [,2] [,3]
## [1,]    1    5    2
## [2,]    0   -3    4
## [3,]    1    1    7

m ^ -1

##      [,1]    [,2]   [,3]
## [1,]    1  0.2000 0.5000
## [2,]  Inf -0.3333 0.2500
## [3,]    1  1.0000 0.1429

(inverse_of_m <- solve(m))

##      [,1] [,2] [,3]
## [1,]  -25  -33   26
## [2,]    4    5   -4
## [3,]    3    4   -3

m %*% inverse_of_m

##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    1    0
## [3,]    0    0    1
```

4.4 小结

- seq 函数及其变种可创建数字序列。
- 向量有长度，且可以使用 length 函数取得其值。
- 可以在创建时指定向量元素的名称，也可以用 names 函数指定。
- 你可以通过指定方括号的索引值来访问向量中的切片。rep 函数能创建重复的向量元素。
- 数组是多维对象，矩阵是二维数组的特例。
- nrow、ncol 和 dim 函数提供了访问数组维度的方法。
- 同样地，rownames、colnames 和 dimnames 能够取得数组中维度的名称。

4.5 知识测试：问题

- 问题 4-1
  你将如何创建一个包含值 0、0.25、0.5、0.75 和 1 的向量？

注 2：qr.solve(m) 和 chol2inv(chol(m)) 提供了另一种矩阵求反的算法，但 solve 函数仍应是首选。

- 问题 4-2

  描述两种命名向量元素的方式。

- 问题 4-3

  向量索引中的四种类型是什么？

- 问题 4-4

  一个 3×4×5 的数组的长度是多少？

- 问题 4-5

  你会用哪个操作符来执行两个矩阵的内积？

4.6 知识测试：练习

- 练习 4-1

  第 `n` 个三角形数表示为 `n * (n + 1) / 2`。创建一个包含前 20 个三角形数的序列。R 有一个内置常数 `letters`，它包含小写的罗马字母。使用前 20 个英文字母来给你刚刚创建的向量命名。选择命名为元音的三角数。[10]

- 练习 4-2

  diag 函数有几种用途，其中之一是以输入向量作为对角线来创建一个方阵。使用序列 10 到 0 到 11（即 11,10,⋯,1,0,1,⋯,11）创建一个 21×21 的矩阵。[5]

- 练习 4-3

  你可通过给 `diag` 函数传递两个额外的参数来指定输出的维度。创建一个主对角线元素都为 1 的 20×21 的矩阵。现在，在此矩阵之上加一行全零元素来创建一个 21×21 的方阵，原来主对角线上的全 1 元素现在全体向下偏移一行。

  创建另一个矩阵，使主对象线上的全 1 元素往上偏移一行。

  把这两个矩阵相加，然后再与 4-2 练习中的答案相加。所得的矩阵被称为 Wilkinson 矩阵。

  `eigen` 函数计算矩阵的特征值和特征向量。计算 Wilkinson 矩阵的特征值。你注意到了什么吗？ [20]

第 5 章

列表和数据框

到目前为止，向量、矩阵和数组所包含的元素的类型都是相同的。列表和数据框是两种特殊的类型，允许我们把不同类型的数据合并到单一变量中。

5.1 本章目标

阅读本章后，你会了解以下内容：

- 如何创建列表和数据框；
- 如何使用 `length`、`names` 和其他函数来检查和操作这些变量类型；
- `NULL` 是什么，何时使用它；
- 递归变量和原子变量之间的区别；
- 列表和数据框的基本操作方法。

5.2 列表

不严格地说，列表是一个向量，其中每个元素的类型可以不同。本节谈一谈如何创建、索引和操作列表。

5.2.1 创建列表

列表由 `list` 函数创建，且能像 `c` 函数那样指定内容。你只需简单地用逗号分隔每个参数即可指定列表中的内容。列表中的元素变量的类型不限，可以是向量、矩阵，甚至函数：

```
(a_list <- list(
  c(1, 1, 2, 5, 14, 42),    #See http://oeis.org/A000108
  month.abb,
  matrix(c(3, -8, 1, -3), nrow = 2),
  asin
))

## [[1]]
## [1]  1  1  2  5 14 42
##
## [[2]]
##  [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov"
## [12] "Dec"
##
## [[3]]
##      [,1] [,2]
## [1,]    3    1
## [2,]   -8   -3
##
## [[4]]
## function (x)  .Primitive("asin")
```

与向量的命名类似，你可以在构造列表时就给元素命名，或在构造之后使用 names 函数命名：

```
names(a_list) <- c("catalan", "months", "involutary", "arcsin")
a_list

## $catalan
## [1]  1  1  2  5 14 42
##
## $months
##  [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov"
## [12] "Dec"
##
## $involutary
##      [,1] [,2]
## [1,]    3    1
## [2,]   -8   -3
##
## $arcsin
## function (x)  .Primitive("asin")

(the_same_list <- list(
  catalan    = c(1, 1, 2, 5, 14, 42),
  months     = month.abb,
  involutary = matrix(c(3, -8, 1, -3), nrow = 2),
  arcsin     = asin
))

## $catalan
## [1]  1  1  2  5 14 42
##
## $months
##  [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov"
```

```
## [12] "Dec"
##
## $involutary
##      [,1] [,2]
## [1,]    3    1
## [2,]   -8   -3
##
## $arcsin
## function (x)  .Primitive("asin")
```

尽管这并非是强制性的，但你在给元素命名时最好使用有效的变量名。

你甚至可以把列表作为一个列表的元素：

```
(main_list <- list(
  middle_list          = list(
    element_in_middle_list = diag(3),
    inner_list             = list(
      element_in_inner_list         = pi ^ 1:4,
      another_element_in_inner_list = "a"
    )
  ),
  element_in_main_list = log10(1:10)
))

## $middle_list
## $middle_list$element_in_middle_list
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    1    0
## [3,]    0    0    1
##
## $middle_list$inner_list
## $middle_list$inner_list$element_in_inner_list
## [1] 3.142
##
## $middle_list$inner_list$another_element_in_inner_list
## [1] "a"
##
##
##
## $element_in_main_list
##  [1] 0.0000 0.3010 0.4771 0.6021 0.6990 0.7782 0.8451 0.9031 0.9542 1.0000
```

从理论上讲，你可以一直这样嵌套下去。实际上，一旦嵌套列表达到成千上万层（确切的数字在不同机器上有所不同），当前版本的 R 会抛出一个错误。幸好，在现实中这不成问题，因为嵌套超过三、四个的代码已是极少见了。

5.2.2 原子变量和递归变量

由于列表具有这种把其他列表包含在内的能力，它被认为是*递归*变量。与之相对，向量、矩阵和数组则是*原子*变量（变量或是递归的，或是原子的，从不两者兼具。附录 A 的表解

释了哪些变量类型是原子的，哪些是递归的）。 函数 is.recursive 和 is.atomic 可测试它们的类型：

```
is.atomic(list())
## [1] FALSE
is.recursive(list())
## [1] TRUE
is.atomic(numeric())
## [1] TRUE
is.recursive(numeric())
## [1] FALSE
```

5.2.3　列表的维度和算术运算

列表与向量一样也有长度，其长度是它顶层元素的数目：

```
length(a_list)
## [1] 4
length(main_list)     # 不包括嵌套列表的长度
## [1] 2
```

邮

电

仍与向量类似但异于矩阵的是，列表没有维度。因此，把 dim 函数作用于列表上会返回 NULL：

```
dim(a_list)
## NULL
```

nrow、NROW 及相应的列函数作用于列表时，与作用于矢量上的原理基本相同：

```
nrow(a_list)
## NULL
ncol(a_list)
## NULL
NROW(a_list)
## [1] 4
NCOL(a_list)
## [1] 1
```

与向量不同的是，算术运算对列表不起作用。由于每个元素的类型可以不同，将两个列表相加或相乘没有任何意义。然而，如果两个列表中的元素类型相同，则可对它们进行算术运算。在这种情况下，一般的算术运算法都适用。例如：

```
l1 <- list(1:5)
l2 <- list(6:10)
l1[[1]] + l2[[1]]
```

```
## [1] 7 9 11 13 15
```

更常见的是，你可能要对列表中的每一个元素执行算术运算（或其他一些操作）。这需要使用循环，第 8 章中详细介绍。

5.2.4　索引列表

考虑这个测试列表：

```
l <- list(
  first  = 1,
  second = 2,
  third  = list(
    alpha = 3.1,
    beta  = 3.2
  )
)
```

与向量类似，我们可通过方括号 []、正或负的下标数字、元素名称或逻辑索引这四种方法访问列表中的元素。以下四行代码的结果相同：

```
l[1:2]
```

```
## $first
## [1] 1
##
## $second
## [1] 2
```

```
l[-3]
```

```
## $first
## [1] 1
##
## $second
## [1] 2
```

```
l[c("first", "second")]
```

```
## $first
## [1] 1
##
## $second
## [1] 2
```

```
l[c(TRUE, TRUE, FALSE)]

## $first
## [1] 1
##
## $second
## [1] 2
```

这些索引操作的结果催生了另一个列表。有时，我们要访问的是列表元素中的内容。有两个操作符能帮助我们做到这一点：可给双方括号 `[[ ]]` 传入正整数，它代表返回的索引下标，或指定该元素的名称字符串：

```
l[[1]]

## [1] 1

l[["first"]]

## [1] 1
```

如果输入的是一个列表，`is.list` 函数将返回 `TRUE`，否则将返回 `FALSE`。作为比较，观察以下两个索引操作符：

```
is.list(l[1])

## [1] TRUE

is.list(l[[1]])

## [1] FALSE
```

对于列表中的命名元素，我们也可以使用美元符号运算符 `$`。这与向双方括号传入一个命名字符串的工作方式几乎一样，且另有两个好处。首先，许多 IDE 能自动补全名字（在 R 的 GUI 下，按下 Tab 键可实现此功能）。其次，R 能接受部分匹配的元素名称：

```
l$first

## [1] 1

l$f       # 部分匹配将 "f" 解释为 "first"

## [1] 1
```

可以通过嵌套方括号或传入向量来访问嵌套元素，尽管后者不常用且往往难以阅读：

```
l[["third"]]["beta"]

## $beta
## [1] 3.2

l[["third"]][["beta"]]
```

```
## [1] 3.2
```

```
l[[c("third", "beta")]]
```

```
## [1] 3.2
```

当尝试访问不存在的元素列表时，根据你所使用的索引类型，行为会有所不同。在下例中，回想一下列表 l，它只有三个元素。

如果我们使用单方括号的索引，那么将返回只带一个 NULL 元素的列表（且名称为 NA，如原列表有名字的话）。与之相比，对于一个无效的向量索引，其返回值将是 NA：

```
l[c(4, 2, 5)]
```

```
## $<NA>
## NULL
##
## $second
## [1] 2
##
## $<NA>
## NULL
```

```
l[c("fourth", "second", "fifth")]
```

```
## $<NA>
## NULL
##
## $second
## [1] 2
##
## $<NA>
## NULL
```

无论是以双方括号还是美元符号来访问元素的内容，如果名字错误，都将返回 NULL：

```
l[["fourth"]]
```

```
## NULL
```

```
l$fourth
```

```
## NULL
```

最后，如果以错误的数字索引访问元素内容，则会抛出一个错误，说明下标越界。对于这种不一致的行为，你只能接受，而最好的防范措施是确保在使用前对索引作相应的检查：

```
l[[4]]  # 这里会抛出一个错误
```

5.2.5 向量和列表之间的转换

向量可使用函数 as.list 函数来转换成列表。所创建的可见列表与向量中元素的值一一对应：

```
busy_beaver <- c(1, 6, 21, 107) # 请查看 http://oeis.org/A060843
as.list(busy_beaver)

## [[1]]
## [1] 1
##
## [[2]]
## [1] 6
##
## [[3]]
## [1] 21
##
## [[4]]
## [1] 107
```

如果列表中每个元素都是标量值，则亦可使用之前出现过的函数（as.numeric、as.character 等）将其转换成向量：

```
as.numeric(list(1, 6, 21, 107))
## [1] 1 6 21 107
```

如果列表中包含非标量元素，这种方法将不起作用。这真的是一个问题，因为列表既能存储不同类型的数据，也能存储相同类型的数据，但后者不是存储成矩阵的形式：

```
(prime_factors <- list( two = 2,
   three = 3,
   four = c(2,  2),
   five = 5,
   six = c(2,   3),
   seven = 7,
   eight = c(2,  2,  2),
   nine = c(3,  3),
   ten = c(2,  5)
))

## $two
## [1] 2
##
## $three
## [1] 3
##
## $four
## [1] 2 2
##
## $five
## [1] 5
##
## $six
## [1] 2 3
##
## $seven
## [1] 7
##
```

```
## $eight
## [1] 2 2 2
##
## $nine
## [1] 3 3
##
## $ten
## [1] 2 5
```

对于这样的列表，可用 unlist 函数将其转换为向量（对于混合类型的列表，有时技术上可行，但没什么用）：

```
unlist(prime_factors)
```

```
##    two  three  four1  four2   five   six1   six2  seven eight1 eight2
##      2      3      2      2      5      2      3      7      2      2
## eight3  nine1  nine2   ten1   ten2
##      2      3      3      2      5
```

5.2.6 组合列表

c 函数既能用于拼接向量，亦能用于拼接列表：

```
c(list(a = 1, b = 2), list(3))
```

```
## $a
## [1] 1
##
## $b
## [1] 2
##
## [[3]]
## [1] 3
```

如果我们用它来拼接列表和向量，向量在拼接之前将被转换为列表（就像调用了 as.list 函数）：

```
c(list(a = 1, b = 2), 3)
```

```
## $a
## [1] 1
##
## $b
## [1] 2
##
## [[3]]
## [1] 3
```

亦可对列表使用 cbind 和 rbind 函数，但结果中出现的对象相当奇怪。它们或是含有可能为非标量元素的矩阵，或是有维度的列表，这取决于你看它们的方式：

```
(matrix_list_hybrid <- cbind(
  list(a = 1, b = 2),
  list(c = 3, list(d = 4))
))

##   [,1] [,2]
## a 1    3
## b 2    List,1

str(matrix_list_hybrid)

## List of 4
##  $ : num 1
##  $ : num 2
##  $ : num 3
##  $ :List of 1
##   ..$ d: num 4
##  - attr(*, "dim")= int [1:2] 2 2
##  - attr(*, "dimnames")=List of 2
##   ..$ : chr [1:2] "a" "b"
##   ..$ : NULL
```

在此，你应避免常用 `cbind` 和 `rbind`，或许最好完全不用它们。这再次证明 R 有点过于灵活和包容了，否则它会直接抛出一个错误表明你干了蠢事。

5.3 NULL

`NULL` 是个特殊值，它表示一个空的变量。它最常用于列表中，不过也会出现在数据框和函数参数中，这些将在后文中讨论。

在创建列表时，你可能会想指定一个元素，表明它必须存在但没有赋值。例如，下表按月份列出了英国在 2013 年的银行假期 [1]。有几个月没有公假日，我们用 `NULL` 表示：

```
(uk_bank_holidays_2013 <- list(
  Jan = "New Year's Day",
  Feb = NULL,
  Mar = "Good Friday",
  Apr = "Easter Monday",
  May = c("Early May Bank Holiday",   "Spring Bank Holiday"),
  Jun = NULL,
  Jul = NULL,
  Aug = "Summer Bank Holiday",
  Sep = NULL,
  Oct = NULL,
  Nov = NULL,
  Dec = c("Christmas Day", "Boxing Day")
 ))
```

注 1：银行假期指公众假期。

```
## $Jan
## [1] "New Year's Day"
##
## $Feb
## NULL
##
## $Mar
## [1] "Good Friday"
##
## $Apr
## [1] "Easter Monday"
##
## $May
## [1] "Early May Bank Holiday" "Spring Bank Holiday"
##
## $Jun
## NULL
##
## $Jul
## NULL
##
## $Aug
## [1] "Summer Bank Holiday"
##
## $Sep
## NULL
##
## $Oct
## NULL
##
## $Nov
## NULL
##
## $Dec
## [1] "Christmas Day" "Boxing Day"
```

理解 NULL 和特殊的缺失值 NA 之间的区别非常重要。最大的区别是，NA 是一个标量值，而 NULL 不会占用任何空间——它的长度为零：

```
length(NULL)
```

```
## [1] 0
```

```
length(NA)
```

```
## [1] 1
```

你可以使用函数 is.null 测试变量是否为 NULL 值。缺失值不是 NULL：

```
is.null(NULL)
```

```
## [1] TRUE
```

```
is.null(NA)
```

```
## [1] FALSE
```

反过来的测试没有太大的意义。因为 `NULL` 的长度为零，我们无法测试它是否是缺失的：

```
is.na(NULL)
```

```
## Warning: is.na() applied to non-(list or vector) of type 'NULL'
## logical(0)
```

`NULL` 也可用于删除列表中的元素。把元素设置为 `NULL`（即使它已经是 `NULL`）则会删除它。假设由于某种原因，我们要切换到一个老式的罗马十月式日历，它没有一月和二月：

```
uk_bank_holidays_2013$Jan <- NULL
uk_bank_holidays_2013$Feb <- NULL
uk_bank_holidays_2013
```

```
## $Mar
## [1] "Good Friday"
##
## $Apr
## [1] "Easter Monday"
##
## $May
## [1] "Early May Bank Holiday" "Spring Bank Holiday"
##
## $Jun
## NULL
##
## $Jul
## NULL
##
## $Aug
## [1] "Summer Bank Holiday"
##
## $Sep
## NULL
##
## $Oct
## NULL
##
## $Nov
## NULL
##
## $Dec
## [1] "Christmas Day" "Boxing Day"
```

要将现有元素设置为 `NULL` 值，我们不能简单地为其分配 `NULL`，因为这将删去元素。而是，它必须使用 `list(NULL)` 来设置。现在，假设英国政府取消了夏季银行假期：

```
uk_bank_holidays_2013["Aug"] <- list(NULL)
uk_bank_holidays_2013
```

```
## $Mar
## [1] "Good Friday"
##
## $Apr
## [1] "Easter Monday"
##
## $May
## [1] "Early May Bank Holiday" "Spring Bank Holiday"
##
## $Jun
## NULL
##
## $Jul
## NULL
##
## $Aug
## NULL
##
## $Sep
## NULL
##
## $Oct
## NULL
##
## $Nov
## NULL
##
## $Dec
## [1] "Christmas Day" "Boxing Day"
```

5.4 成对列表

R 有另外一种形式的列表：*成对列表*（Pairlists）。成对列表只在内部使用，用于将参数传递到函数中，但你一般不会主动使用到它们。而它唯一可能被显式调用的情形是在使用 `formals` 时。该函数将返回一个函数参数的成对列表。

在使用帮助页面查找标准偏差函数 `?sd` 时，我们了解到它需要两个参数，向量 `x` 和逻辑值 `na.rm`，后者的默认值是 `FALSE`：

```
(arguments_of_sd <- formals(sd))
```

```
## $x
##
##
## $na.rm
```

注 2：R 在其基础环境中，还以成对列表的变量形式存储了一些称为 `.Options` 的全局设置。你不能直接访问这个变量，而须使用 `option` 函数来返回其列表。

```
## [1] FALSE
```

```
class(arguments_of_sd)
```

```
## [1] "pairlist"
```

在实际使用中，成对列表与列表几乎一样。唯一的区别是，长度为零的成对列表为 NULL，而长度为零的列表是一个空列表：

```
pairlist()
```

```
## NULL
```

```
list()
```

```
## list()
```

5.5 数据框

数据框用于存储类似电子表格的数据。它们既可被看作是每列可存储不同数据类型的矩阵，或是非嵌套的列表，其中每个元素具有相同的长度。

5.5.1 创建数据框

我们用 data.frame 函数创建数据框：

```
(a_data_frame <- data.frame(
  x = letters[1:5],
  y = rnorm(5),
  z = runif(5) > 0.5
))
```

```
##   x        y    z
## 1 a  0.17581 TRUE
## 2 b  0.06894 TRUE
## 3 c  0.74217 TRUE
## 4 d  0.72816 TRUE
## 5 e -0.28940 TRUE
```

```
class(a_data_frame)
```

```
## [1] "data.frame"
```

请注意，每列的类型可与其他列不同，但在同一列中的元素类型必须相同。还要注意的是，对象的类名是 data.frame，中间有一个点，而非空字符。

在此例中，行自动从一到五编号。如果输入的任何向量有名称，那么行名称就取自第一个向量名称。例如，如果 y 列有命名，那么数据框中每行的名字将以 y 列的向量名命名：

```
y <- rnorm(5)
names(y) <- month.name[1:5]
data.frame(
  x = letters[1:5],
  y = y,
  z = runif(5) > 0.5
)

##          x       y     z
## January  a -0.9373 FALSE
## February b  0.7314  TRUE
## March    c -0.3030  TRUE
## April    d -1.3307 FALSE
## May      e -0.6857 FALSE
```

这种命名规则可通过给 `data.frame` 函数传入参数 `row.names = NULL` 覆盖掉：

```
data.frame(
  x = letters[1:5],
  y = y,
  z = runif(5) > 0.5,
  row.names = NULL
)

##   x       y     z
## 1 a -0.9373 FALSE
## 2 b  0.7314 FALSE
## 3 c -0.3030  TRUE
## 4 d -1.3307  TRUE
## 5 e -0.6857 FALSE
```

另外，还可以通过给 `row.names` 传入一个向量来为每行命名。这个向量将被转换为字符 `character`，如果它不是此类型的话：

```
data.frame(
  x = letters[1:5],
  y = y,
  z = runif(5) > 0.5,
  row.names = c("Jackie", "Tito", "Jermaine", "Marlon", "Michael")
)

##          x       y     z
## Jackie   a -0.9373  TRUE
## Tito     b  0.7314 FALSE
## Jermaine c -0.3030  TRUE
## Marlon   d -1.3307 FALSE
## Michael  e -0.6857 FALSE
```

像矩阵一样行的名称可通过使用 `rownames`（或 `row.names`）在后期进行检索或更改。同样地，也可以使用 `colnames` 和 `dimnames` 来分别获取或置列和维度的名称。事实上，几乎所有用于矩阵的函数亦可用在数据框上。例如，`nrow`、`ncol` 和 `dim` 函数的使用与矩阵一样：

```
rownames(a_data_frame)

## [1] "1" "2" "3" "4" "5"

colnames(a_data_frame)

## [1] "x" "y" "z"

dimnames(a_data_frame)

## [[1]]
## [1] "1" "2" "3" "4" "5"
##
## [[2]]
## [1] "x" "y" "z"

nrow(a_data_frame)

## [1] 5

ncol(a_data_frame)

## [1] 3

dim(a_data_frame)

## [1] 5 3
```

需注意两个奇怪之处。首先，length 将返回与 ncol 相同的值，而不是数据框元素的总数。同样地，names 将返回与 colnames 相同的值。为了使代码易于理解，建议你避开这两个函数，而使用 ncol 和 colnames：

```
length(a_data_frame)

## [1] 3

names(a_data_frame)

## [1] "x" "y" "z"
```

可以使用不同长度的向量来创建数据框，只要长度较短的向量能刚好循环至总长度。从技术上讲，所有向量长度的最小公倍数必须与最长向量的长度相等：

```
data.frame(       # 长度 1、2 和 4 是 OK 的
  x = 1,          # 循环 4 次
  y = 2:3,        # 循环 2 次
  z = 4:7         # 最长的输入，不需要循环
)

##   x y z
## 1 1 2 4
## 2 1 3 5
## 3 1 2 6
## 4 1 3 7
```

如果长度不兼容，将引发错误：

```
data.frame(      # 长度 1、2 和 3 将导致错误
  x = 1,         # 最小公倍数是 6，比 3 大
  y = 2:3,
  z = 4:6
)
```

创建数据框时的另一个要点为：在默认情况下，列名必须是唯一且有效的变量名称。此功能可通过给 data.frame 传入 check.names = FALSE 来关闭：

```
data.frame(
  "A column"    = letters[1:5],
  "!@#$%^&*()" = rnorm(5),
  "..."         = runif(5) > 0.5,
  check.names  = FALSE
)

##   A column !@#$%^&*()   ...
## 1        a    0.32940  TRUE
## 2        b   -1.81969  TRUE
## 3        c    0.22951 FALSE
## 4        d   -0.06705  TRUE
## 5        e   -1.58005  TRUE
```

一般情况下，使用非标准的列名并不好。复制列名则更糟糕，因为一旦你使用子集，它会导致难以发现的错误。另外，如果关闭列名检查，你将陷于险境。

5.5.2 索引数据框

有很多种不同的方式可用于索引数据框。首先，与矩阵索引的方式一样，可使用四种不同的向量索引（正整数、负整数、逻辑值和字符）。以下的命令将选择数据框中前两列的第二个和第三个元素：

```
a_data_frame[2:3, -3]

##   x       y
## 2 b 0.06894
## 3 c 0.74217

a_data_frame[c(FALSE, TRUE, TRUE, FALSE, FALSE), c("x", "y")]

##   x       y
## 2 b 0.06894
## 3 c 0.74217
```

因为选择了一个以上的列，所以得到的子集也是一个数据框。如果只选择一个列，其结果将被简化为一个向量：

```
class(a_data_frame[2:3, -3])
```

```
## [1] "data.frame"

class(a_data_frame[2:3, 1])

## [1] "factor"
```

如果我们只需选择一个列，那么也可以使用列表样式的索引（带有正整数或名称的双方括号，或者带有名称的美元符号运算符）。以下命令将选择第一列中的第二个和第三个元素：

```
a_data_frame$x[2:3]

## [1] b c
## Levels: a b c d e

a_data_frame[[1]][2:3]

## [1] b c
## Levels: a b c d e

a_data_frame[["x"]][2:3]

## [1] b c
## Levels: a b c d e
```

如果我们需要给列加上条件来得到一个数据框子集，使用的语法会有点冗长，而 `subset` 函数能做同样的事情且更简洁。`subset` 需要三个参数：一个数据框，一个行的条件逻辑向量，以及一个需要保留的名字向量（如果最后这个参数被省略了，那么将保留所有列）。`subset` 的过人之处在于它使用了特殊的估算技巧，以避免多余的操作：你无需输入 `a_data_frame$y` 以访问 `a_data_frame` 的第 `y` 列，因为它已经知道要看哪个数据框，所以你只需键入 `y`。同样地，选择列时，你无需附上引号中列的名称，而是可以直接键入名称。在下例中，记得 `|` 是逻辑或操作符：

```
a_data_frame[a_data_frame$y > 0 | a_data_frame$z, "x"]

## [1] a b c d e
## Levels: a b c d e

subset(a_data_frame, y > 0 | z, x)

##   x
## 1 a
## 2 b
## 3 c
## 4 d
## 5 e
```

5.5.3 基本数据框操作

像矩阵一样，数据框可使用 t 函数进行转置。但在此过程中，所有的列（它们即将变成行）

将被转换为相同的类型，然后将变成一个矩阵：

```
t(a_data_frame)
```

```
##   [,1]         [,2]         [,3]         [,4]         [,5]
## x "a"          "b"          "c"          "d"          "e"
## y " 0.17581"   " 0.06894"   " 0.74217"   " 0.72816"   "-0.28940"
## z "TRUE"       "TRUE"       "TRUE"       "TRUE"       "TRUE"
```

如果两个数据框的大小一致，也可使用 cbind 和 rbind 把它们连接（join）起来。rbind 能智能地对列重新排序以匹配。然而，因为 cbind 不会对列名作重复性检查，使用时要格外小心：

```
another_data_frame <- data.frame(    # 与 a_data_frame 有相同的 cols，不过次序不同
  z = rlnorm(5),                     # 对数分布的数
  y = sample(5),                     # 1 到 5 随机排列的数
  x = letters[3:7]
)
rbind(a_data_frame, another_data_frame)
```

```
##    x        y      z
## 1  a  0.17581 1.0000
## 2  b  0.06894 1.0000
## 3  c  0.74217 1.0000
## 4  d  0.72816 1.0000
## 5  e -0.28940 1.0000
## 6  c  1.00000 0.8714
## 7  d  3.00000 0.2432
## 8  e  5.00000 2.3498
## 9  f  4.00000 2.0263
## 10 g  2.00000 1.7145
```

```
cbind(a_data_frame, another_data_frame)
```

```
##   x        y    z      z y x
## 1 a  0.17581 TRUE 0.8714 1 c
## 2 b  0.06894 TRUE 0.2432 3 d
## 3 c  0.74217 TRUE 2.3498 4 e
## 4 d  0.72816 TRUE 2.0263 5 f
## 5 e -0.28940 TRUE 1.7145 2 g
```

当两个数据框有相同的列时，可使用 merge 函数合并。merge 为数据库风格的连接提供了多种选择。当要连接两个数据框时，你需要指定包含键值的列以作匹配。默认情况下，merge 函数会使用两个数据框中所有共同的列，但通常你会想用一个共享 ID 列。在下例中，我们将通过 by 参数指定 x 为我们所要包含的 ID：

```
merge(a_data_frame, another_data_frame, by = "x")
```

```
##   x     y.x   z.x     z.y y.y
## 1 c  0.7422  TRUE  0.8714   1
## 2 d  0.7282  TRUE  0.2432   3
## 3 e -0.2894  TRUE  2.3498   5
```

```
merge(a_data_frame, another_data_frame, by = "x", all = TRUE)
```

```
##   x       y.x   z.x      z.y y.y
## 1 a  0.17581  TRUE       NA  NA
## 2 b  0.06894  TRUE       NA  NA
## 3 c  0.74217  TRUE   0.8714   1
## 4 d  0.72816  TRUE   0.2432   3
## 5 e -0.28940  TRUE   2.3498   5
## 6 f       NA    NA   2.0263   4
## 7 g       NA    NA   1.7145   2
```

如果数据框中只包含数值，那么 `colSums` 和 `colMeans` 函数可分别用于计算每列的总和和平均值。同样地，`rowSums` 和 `rowMeans` 将计算每一行的总和及平均值：

```
colSums(a_data_frame[, 2:3])
```

```
##     y     z
## 1.426 5.000
```

```
colMeans(a_data_frame[, 2:3])
```

```
##      y      z
## 0.2851 1.0000
```

操作数据框是一个很大的话题，我们将在第 13 章深入探讨。

5.6 小结

- 列表中的每个元素中可包含大小和类型都不同的变量。
- 列表是递归变量，因为它可以包含其他列表。
- 使用 `[ ]`、`[[ ]]` 或 `$` 来索引列表。
- `NULL` 是特殊值，用于创建“空”的列表元素。
- 数据框能存储类似电子表格的数据。
- 数据框有一些类似矩阵（矩形的）和列表（不同的列可以包含不同类型的变量）的属性。
- 数据框可以被索引，正如矩阵或列表一样。
- `merge` 能让你对数据框进行类似数据库风格的连接操作。

5.7 知识测试：问题

- 问题 5-1

  以下列表的长度是多少？

  ```
  list(alpha = 1, list(beta = 2, gamma = 3, delta = 4), eta = NULL)
  ```

  ```
  ## $alpha
  ```

```
## [1] 1
##
## [[2]]
## [[2]]$beta
## [1] 2
##
## [[2]]$gamma
## [1] 3
##
## [[2]]$delta
## [1] 4
##
##
## $eta
## NULL
```

- 问题 5-2

  你会在哪里找到成对列表？

- 问题 5-3

  尽可能多地说出的几种创建数据框子集的方法。

- 问题 5-4

  如何创建一个数据框，使得它的列名既非唯一又非有效？

- 问题 5-5

  你会使用哪个函数将一个数据框追加到另一个之后？

5.8 知识测试：练习

- 练习 5-1

  创建一个列表变量，它的第一个元素包含所有从 0 到 9 的平方数，第二个元素为 10 至 19 之内的所有平方数，依此类推，最后一个元素为 90 到 99 之内的平方数。不是平方数的元素也应该被包含在内！ [10]

- 练习 5-2

  R 有几个内置的数据集，其中包括著名的 [3]、由安德森和费舍尔在 20 世纪 30 年代收集和分析的 iris（指鸢尾花，而不是虹膜）数据。输入 `iris` 即可看到数据集。创建一个新的数据框，它由 iris 数据集的数值列组成；计算各列的平均值。[5]

- 练习 5-3

  `beaver1` 和 `beaver2` 数据集包含两个海狸的体温数据。为 `beaver1` 数据集添加一列名为 `id` 的列，其值全部为 1。同样，也为 `beaver2` 添加一个 `id` 列，值全为 2。垂直拼接两个数据框，并且找到所有活跃着的海狸的子集。[10]

注 3：根据一些“著名”的定义。

第 6 章

环境和函数

我们已经尝试了 R 中的多种函数。在本章中，你将了解函数是什么，以及如何编写你自己的函数。在此之前，先来看看用于存储变量的环境。

6.1 本章目标

阅读本章后，你会了解以下内容：

- 环境是什么，如何创建它；
- 如何创建、访问和列出环境内的变量；
- 函数的各个组成部分；
- 编写自己的函数；
- 变量的作用域。

6.2 环境

我们所创建的所有变量都需要存储在某处，即环境。环境本身也是另一种类型的变量，我们可以像对待其他变量一样随意地分配和操作它们，并将其以参数的形式传递到函数中。它们与列表密切相关，也能用于存储不同类型的变量。事实上，大部分用于列表的语法也同样适用于环境，而且我们也可以强制地把列表强制当作环境使用（反之亦然）。

通常情况下，你不需要直接与环境打交道。例如，你在命令提示符下分配一个变量，它会自动进入全局环境（也称为用户工作区）中。当你调用函数时，将会自动创建一个环境，用于存储与此函数相关的变量。不过，当你要了解变量的作用域或调试代码以检查调用栈

时，掌握环境的一些基础知识将有所帮助。

有些烦人的是，环境的创建不是使用 environment 函数（该函数将返回包含的特定函数的环境），而是使用 new.env 函数：

```
an_environment<-new.env()
```

向环境中分配变量的方式与列表完全相同。可以使用双方括号或美元符号运算符。和列表一样，环境变量的类型和大小可以不同：

```
an_environment[["pythag"]]<-c(12, 15, 20, 21)   # 请查看 http://oeis.org/A156683
an_environment$root<-polyroot(c(6, -5, 1))
```

2.3 节中见到 assign 函数还有一个可选的环境参数，用于指定变量的存储位置：

```
assign(
  "moonday",
  weekdays(as.Date("1969/07/20")),
  an_environment
)
```

检索变量的方式也是如此：你可以使用列表的索引语法，或 assign 的对立函数 get：

```
an_environment[["pythag"]]

## [1] 12 15 20 21

an_environment$root

## [1] 2+0i 3-0i

get("moonday", an_environment)

## [1] "Sunday"
```

可以把环境参数传入 ls 和 ls.str 函数中，列出它的所有内容：

```
ls(envir = an_environment)

## [1] "moonday" "pythag" "root"

ls.str(envir = an_environment)

## moonday: chr "Sunday"
## pythag: num [1:4] 12 15 20 21
## root: cplx [1:2] 2+0i 3-0i
```

可用 exists 函数测试变量是否在环境中：

```
exists("pythag", an_environment)

## [1] TRUE
```

很明显，使用 as.list 和 as. environment 函数能分别实现从环境到列表或相反过程的转换。在后一种情况中，还可以使用 list2env 函数，它在创建环境时更为灵活：

```
# 转换为列表
(a_list<-as.list(an_environment))

## $pythag
## [1] 12 15 20 21
##
## $moonday
## [1] "Sunday"
##
## $root
## [1] 2+0i 3-0i

# 再转换回来。以下两行代码的效果一样。
as.environment(a_list)

## <environment: 0x000000004a6fe290>

list2env(a_list)

## <environment: 0x000000004ad10288>
```

所有的环境都是嵌套的，这意味着它们必须有一个父环境（除了位于顶端的特殊环境——**空环境**以外）。默认情况下，exists 和 get 函数也将在父环境中寻找变量。可通过给它们传入 inherits = FALSE 来改变这种行为，使它们仅在指定的环境中搜索：

```
nested_environment<-new.env(parent=an_environment)
exists("pythag", nested_environment)

## [1] TRUE

exists("pythag", nested_environment, inherits=FALSE)

## [1] FALSE
```

> “框”（frame）这个词几乎可以与“环境”互换（要想了解此术语，请参考随 R 附带的语言定义手册 2.1.10 节）。这意味着工作环境中的一些函数名称里可能有“frame”，最常见的是 parent.frame。

下列快捷函数可以同时访问全局环境（那些你从命令提示符中分配的变量的存储空间）和基础环境（那些 R 基础包中自带的基础函数和变量）：

```
non_stormers<<-c(3, 7, 8, 13, 17, 18, 21)  # 参见 http://oeis.org/A002312
get("non_stormers", envir=globalenv())

## [1] 3 7 8 13 17 18 21

head(ls(envir=baseenv()), 20)
```

```
##  [1] "-"                 "-.Date"            "-.POSIXt"
##  [4] "!"                 "!.hexmode"         "!.octmode"
##  [7] "!="                "$"                 "$.data.frame"
## [10] "$.DLLInfo"         "$.package_version" "$<-"
## [13] "$<-.data.frame"    "%%"                "%*%"
## [16] "%/%"               "%in%"              "%o%"
## [19] "%x%"               "&"
```

在另外两种情况下也可能遇到环境。首先，调用函数时，函数所定义的所有变量都被存储在属于该函数的环境中（函数及其环境有时称为闭包）。其次，加载包时，包中的函数将存储于其搜索路径的环境中。这将在第 10 章中讨论。

6.3　函数

大多数的变量类型仅用于存储数据，而函数能够让我们和数据一起工作，它们是“动词”而非“名词”。和环境类似，它们只是另一种数据类型，我们可以分配、操纵，甚至将它传递给其他函数的数据类型。

6.3.1　创建和调用函数

为了更好地了解函数，我们来看看它的组成。

键入一个函数的名称，将显示其运行的代码。以下是 rt 函数，该函数将生成基于 T 分布的随机数[1]：

```
rt

## function (n, df, ncp)
## {
##     if (missing(ncp))
##         .External(C_rt, n, df)
##     else rnorm(n, ncp)/sqrt(rchisq(n, df)/df)
## }
## <bytecode: 0x0000000019738e10>
## <environment: namespace:stats>
```

如你所见，rt 需要三个输入参数：n 是要产生的随机数的数目，df 是自由度值，ncp 是一个可选的非中心参数。从技术上来说，三个参数 n、df 和 ncp 是 rt 函数的形式参数（formal argument）。当你调用该函数并给它传递值时，这些值被称为参数。

> 参数和形式参数之间的差异不是很重要，因此接下来本书没有区分这两个概念。

注 1：如果定义是类似于 UseMethod（"my_function"）或 standardGeneric（"my_function"）的一行内容，请参阅 16.7 节。如果 R 抱怨找不到对象，尝试 getAnywhere（my_function）。

在大括号之间，你可以看到函数体内代码行。它们就是每次调用 rt 时要执行的代码。

请注意，这里有没有显式的“return”关键字声明应该从函数返回哪个值。在 R 中，函数中计算的最后一个值将自动返回。以 rt 为例，如果 ncp 参数被省略，将会调用 C 代码生成随机数并返回。否则，该函数会调用 rnorm、rchisq 和 sqrt 函数计算并返回值。

要创建我们自己的函数，只需像其他任何变量一样为它赋值。举一个例子，创建一个函数来计算直角三角形斜边的长度（为简单起见，我们将使用一般的算法。但不要在实际项目中使用这些代码，因为它们在数字很大或很小时，这种算法并不适用）：

```
hypotenuse <- function(x, y)
{
  sqrt(x ^ 2 + y ^ 2)
}
```

这里，hypotenuse 是我们正在创建的函数，x 和 y 是它的参数（形参），在大括号中的内容是函数体。

事实上，因为函数体只有一行代码，可省略大括号：

```
hypotenuse <- function(x, y) sqrt(x ^ 2 + y ^ 2)  # 和之前一样
```

R 对于代码如何使用空白符很宽容，所以“一行代码”可以延伸到多行。没有使用大括号的大量代码也是一条语句（statement）。语句的确切定义涉及技术术语，但从实用角度看，它就是在执行前你可在命令行中键入的代码量。

现在，可以使用以下任意一种方式来调用这个函数：

```
hypotenuse(3, 4)

## [1] 5

hypotenuse(y = 24, x = 7)

## [1] 25
```

当我们调用函数时，如果不命名参数，则 R 将按位置匹配它们。以 hypotenuse(3, 4) 为例：3 是第一个参数，因此它对应 x；4 是第二个参数，因此它对应 y。

如果要改变我们传递参数的顺序，或省略其中一些，则可传入命名参数。以 hypotenuse(y = 24, x = 7) 为例，虽然传递变量的顺序是“错误”的，但 R 仍能正确地判断出哪个变量应被映射到 x，哪个应被映射到 y。

它对于计算斜边的函数来说意义不大，但如果有需要，我们可以给 x 和 y 提供默认值。在以下新版本的代码中，如果我们不给函数传递任何值，则 x 会取默认值 5，而 y 会取 12：

```
hypotenuse <- function(x = 5, y = 12)
{
  sqrt(x ^ 2 + y ^ 2)
}
hypotenuse()     # 与 hypotenuse(5, 12) 相等

## [1] 13
```

我们已经见到过 formals 函数，它能取得函数的参数并返回一个（结对）列表。args 函数也能做相同的事，看上去更加可读但编程风格不太友好。formalArgs 函数将返回参数名称的字符向量：

```
formals(hypotenuse)

## $x
## [1] 5
##
## $y
## [1] 12

args(hypotenuse)

## function (x = 5, y = 12)
## NULL

formalArgs(hypotenuse)

## [1] "x" "y"
```

body 函数可用于返回函数体。单独地，它不太有用。但有时我们需要以文本的方式检查它们，例如要查找一个调用了另一函数的函数。对此，可以使用 deparse 函数：

```
(body_of_hypotenuse <- body(hypotenuse))

## {
## sqrt(x^2 + y^2)
## }

deparse(body_of_hypotenuse)

## [1] "{"                   "    sqrt(x^2 + y^2)" "}"
```

函数形参的默认值不仅仅是常数值，我们还可以把任何 R 代码放进去，甚至使用其他形参。下面的 normalize 函数将缩放一个向量。默认情况下，参数 m 和 s 是第一个参数的平均值和标准偏差，所以返回的向量将包含均值 0 和标准偏差 1：

```
normalize <- function(x, m = mean(x), s = sd(x))
{
  (x - m) / s
}
normalized <- normalize(c(1, 3, 6, 10, 15))
```

```
mean(normalized)     # 几乎是 0!
```

```
## [1] -5.573e-18
```

```
sd(normalized)
```

```
## [1] 1
```

不过，normalize 函数有一个小问题，如果 x 的某些元素没有给出，我们将看到：

```
normalize(c(1, 3, 6, 10, NA))
```

```
## [1] NA NA NA NA NA
```

如果向量的所有元素都没有给出，那么在默认情况下，mean 和 sd 都将返回 NA。因此，normalize 函数的返回值中的所有元素都是 NA 值。如果有这样的选项：只有输入值是 NA 时才返回 NA，那可能更好。mean 和 sd 都有一个参数 na.rm，它能让我们删除计算之前的任何缺失值。为了避免所有的 NA 值，我们可以在 normalize 中包含这个参数：

```
normalize <- function(x, m = mean(x, na.rm=na.rm),
  s=sd(x, na.rm=na.rm), na.rm=FALSE)
{
  (x - m) / s
}
normalize(c(1, 3, 6, 10, NA))
```

```
## [1] NA NA NA NA NA
```

```
normalize(c(1, 3, 6, 10, NA), na.rm=TRUE)
```

```
## [1] -1.0215 -0.5108 0.2554 1.2769 NA
```

此方法可行，但语法有点笨拙。为了避免输入那些实际上没有被函数用到的参数名（na.rm 只被传递到 mean 和 sd 中），R 提供了一个特殊参数 ...，它包含了所有不能被位置或名称匹配的参数：

```
normalize<-function(x, m=mean(x, ...), s=sd(x, ...), ...)
{
  (x-m)/s
}
normalize(c(1, 3, 6, 10, NA))
```

```
## [1] NA NA NA NA NA
```

```
normalize(c(1, 3, 6, 10, NA), na.rm=TRUE)
```

```
## [1] -1.0215 -0.5108 0.2554 1.2769 NA
```

现在，在 normalize(c(1, 3, 6, 10, NA), na.rm=TRUE) 的调用里，参数 na.rm 并不能匹配 normalize 的任何形参，因为它不是 x、m 或 s。这意味着它被存储在 normalize 的参

数 ... 里。当我们评估 m 时，表达式 mean(x, ...) 现在为 mean(x, na.rm=TRUE)。

如果你对此还不太清楚，不用担心。大多数时候，它的工作原理是一个我们无需操心的高级话题。当下，你只需了解 ... 能将参数传递给子函数。

6.3.2 向其他函数传递和接收函数

函数可以像其他变量类型一样地使用，我们可将之作为其他函数的参数，并且从函数中返回。一个常见的，把其他函数当成参数的例子是 do.call。此函数提供了一种调用其他函数的替代语法，让我们可以像列表一样传递参数，而不是逐次传递：

```
do.call(hypotenuse, list(x = 3, y = 4))     # 和 hypotenuse(3, 4) 一样

## [1] 5
```

也许最常见的案例为 do.call 与 rbind 混用。你可以结合这两个函数，你可以一次拼接多个数据框或矩阵：

```
dfr1 <- data.frame(x = 1:5, y = rt(5, 1))
dfr2 <- data.frame(x = 6:10, y = rf(5, 1, 1))
dfr3 <- data.frame(x = 11:15, y = rbeta(5, 1, 1))
do.call(rbind, list(dfr1, dfr2, dfr3))     # 和 rbind(dfr1, dfr2, dfr3) 一样

##     x        y
## 1   1  1.10440
## 2   2  0.87931
## 3   3 -1.18288
## 4   4 -1.04847
## 5   5  0.90335
## 6   6  0.27186
## 7   7  2.49953
## 8   8  0.89534
## 9   9  4.21537
## 10 10  0.07751
## 11 11  0.31153
## 12 12  0.29114
## 13 13  0.01079
## 14 14  0.97188
## 15 15  0.53498
```

花一些时间去习惯这种用法是值得的。第 9 章将大量使用 apply 及其衍生函数，把函数传递到其他函数中。

把函数用作参数时，没必要一开始就为它们赋值。以同样的方式将以下函数：

```
menage <- c(1, 0, 0, 1, 2, 13, 80) # 参考 http://oeis.org/A000179
mean(menage)

## [1] 13.86
```

简化为：

```
mean(c(1, 0, 0, 1, 2, 13, 80))

## [1] 13.86
```

我们还可以以匿名方式传递函数：

```
x_plus_y <- function(x, y) x + y
do.call(x_plus_y, list(1:5, 5:1))

## [1] 6 6 6 6 6

# 与下相同
do.call(function(x, y) x + y, list(1:5, 5:1))

## [1] 6 6 6 6 6
```

返回值为函数的情况比较罕见，但它是有效的。ecdf 函数将返回一个向量的经验累积分布函数，如图 6-1 所示：

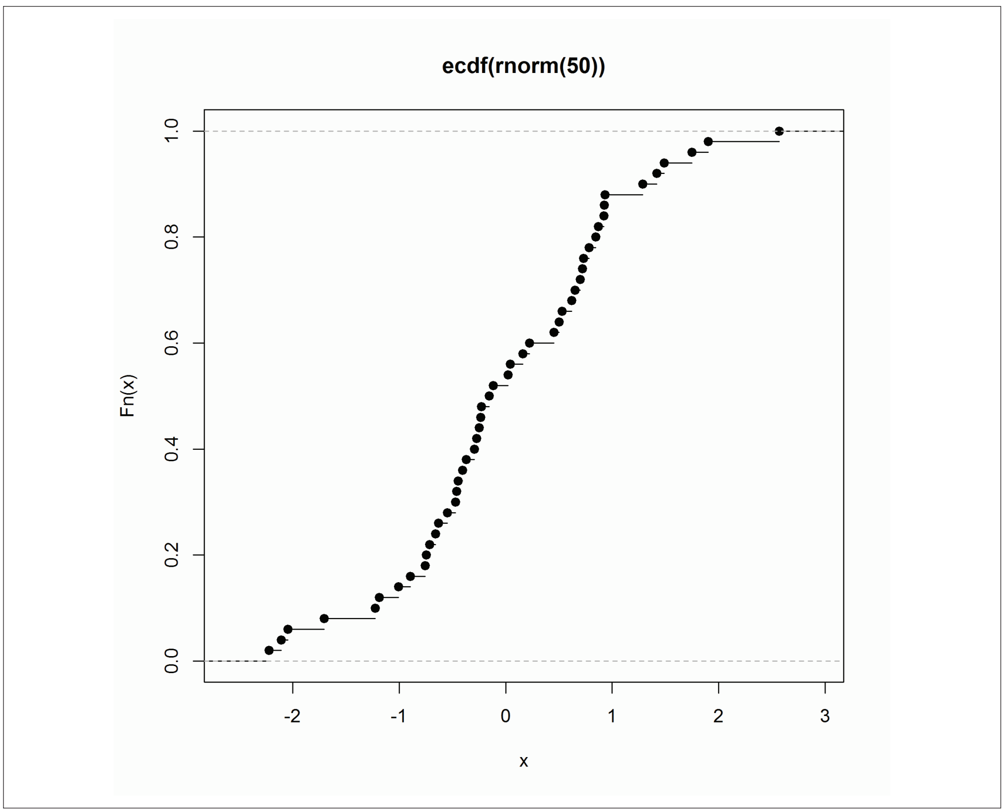


图 6-1：经验累积分布函数

```
(emp_cum_dist_fn <- ecdf(rnorm(50)))

## Empirical CDF
## Call: ecdf(rnorm(50))
##  x[1:50] = -2.2, -2.1,  -2,  ..., 1.9, 2.6

is.function(emp_cum_dist_fn)

## [1] TRUE

plot(emp_cum_dist_fn)
```

6.3.3 变量的作用域

变量的作用域是指你在哪里可以看到此变量。例如，当你在函数内部定义一个变量时，该函数中下面的语句将能访问到该变量。在 R（但不是 S）中，子函数也能访问到这个变量。在下例中，函数 f 的参数为 x，且它将被传递给函数 g。f 还定义了一个变量 y，它的作用域在函数 g 的范围内，这是由于 g 是 f 的子函数。因此，即使没有在 g 里定义 y，下例也能工作：

```
f <- function(x)
{
  y <- 1
  g <- function(x)
  {
    (x + y) / 2     #y 被使用了，但它不是 g 的形式参数
  }
  g(x)
}
f(sqrt(5))          # 这也能工作！ 它神奇地在环境 f 里找到了 y

## [1] 1.618
```

如果我们修改例子，把 g 定义在 f 以外，使 g 不是 f 的子函数，那么例子将抛出一个错误，因为 R 找不到 y：

```
f <- function(x)
{
  y <- 1
  g(x)
}
g <- function(x)
{
  (x + y) / 2
}
f(sqrt(5))

##  January February    March    April      May
##   0.6494   1.4838   0.9665   0.4527   0.7752
```

在 6.2 节中，get 和 exists 函数在其父环境和当前环境下搜索变量。变量作用域的工作方

式与此完全相同：R 将试图在当前的环境变量下寻找变量，如果找不到，则会继续在父环境中搜索，然后再在该环境的父环境中搜索，以此类推，直到达到全局环境。在全局环境中定义的变量在任何地方都可见，这就是它们被称为全局变量的原因。

在第一个例子中，f 的环境是 g 的父环境，因此 y 能被发现。在第二个例子中，g 的父环境是全局环境，它不包含变量 y，所以它会抛出一个错误。

变量可在父环境中被发现，这种作用域系统通常很有用，但它也会带来有缺陷、糟糕和难以维护的代码。考虑下面的函数 h：

```
h <- function(x)
{
  x * y
}
```

这看起来不能工作，因为它只接受一个参数 x，但它的函数体中却使用了两个参数 x 和 y。我们在干净的用户工作区试试：

```
h(9)

##  January February    March    April      May
##   -8.436    6.583   -2.727  -11.976   -6.171
```

到目前为止，我们的直觉是正确的。y 没有被定义，所以该函数将抛出一个错误。现在来看看，如果把 y 定义在用户工作区，会发生什么：

```
y <- 16
h(9)

## [1] 144
```

当 R 在 h 的环境中无法找到一个名为 y 的变量时，它会在 h 的父环境，即定义了 y 的用户工作区（即全局环境）中搜索，然后就能得出正确的乘积。

应该小心使用全局变量，因为它很容易引出错得离谱的代码。在以下被修改过的函数中，h2 和 y 将有一半的几率被随机定义为局部变量。因为 y 被定义于用户工作区中，所以当它执行时，y 是局部变量还是全局变量完全是随机的！

```
h2 <- function(x)
{
  if(runif(1) > 0.5) y <- 12
  x * y
}
```

我们使用 replicate 运行几次来查看其结果：

```
replicate(10, h2(9))
## [1] 144 144 144 108 144 108 108 144 108 108
```

如果用 runif 函数产生的出均匀分布的随机数（在 0 和 1 之间）比 0.5 要大时，局部变量 y 的值被赋值为 12。否则，将使用全局值 16。

我敢肯定你已经注意到，非常容易使代码隐藏有错误。通常来说，更好的做法是显式地向函数传递我们需要的所有变量。

6.4 小结

- 环境能存储变量，它能够被 new.env 函数创建。
- 通常，你可以把环境看作列表。
- 所有的环境都有一个父环境（除了顶部的空环境）。
- 函数由形参和函数体组成。
- 你可以分配和使用函数，像对任何其他的变量类型一样。
- R 将在当前的环境及其父环境中查找变量。

6.5 知识测试：问题

- 问题 6-1
  全局环境的另一个名字是什么？

- 问题 6-2
  如何把列表转换为环境？

- 问题 6-3
  如何将函数的内容打印到控制台上？

- 问题 6-4
  列举三个能够输出函数形参名称的函数。

- 问题 6-5
  do.call 函数会做些什么？

6.6 知识测试：练习

- 练习 6-1
  创建一个名为 multiples_of_pi 的新环境 。将以下变量分配到此环境中：
  (1) two_pi，值为 2 * π，使用双方括号；
  (2) three_pi，值为 3 * π，使用美元符号运算符；
  (3) four_pi，值为 4 * π，使用 assign 函数。
  列出环境的内容，以及它们的值。[10]

- 练习 6-2

  写一个接受一个整数向量的函数（为简单起见，你不用担心输入检查），它将返回一个逻辑向量。其判断逻辑是：如果输入值是偶数则返回 TRUE；奇数则返回 FALSE；非有限值（非有限值指任何能使用 is.finite 函数返回 FALSE 的东西：如 Inf、-Inf、NA 和 NaN）则返回 NA。输入正、负、零和非无限值来检查此函数。[10]

- 练习 6-3

  写一个函数，它接受一个函数作为输入，并返回一个包含两个元素的列表：一个名为 args 的元素，它包含了一个形参的结对列表；一个命为 body 的元素，它包含输入的内容。使用不同的输入测试此函数。[10]

第 7 章 字符串和因子

就像要经常处理数字和逻辑值一样，有时你也必须要处理文本，这在检索或清理数据集时尤为常见。有时，你想把日志文件中的文本转换成有意义的值，或修改数据中的错别字。数据清理将在第 13 章中深入讨论，现在先来学习如何操作字符向量。

因子用于存储类别数据，如性别（男或女），它提供有限的字符串选项。根据上下文的不同，它们使用起来有时像字符向量，有时像整数向量。

7.1 本章目标

阅读本章后，你会了解以下内容：

- 如何从现有的字符串中构造出一个新的字符串；
- 如何格式化数字的打印格式；
- 了解特殊字符，如制表和换行符；
- 如何创建和操作因子。

7.2 字符串

文本数据存储在字符向量中（或字符数组中，虽然这较少见）。重要的是，字符向量中的每个元素都是字符串，而非单独的字符。在 R 中，“字符串”是个常用的非正式术语，因为正式的“字符向量元素”读起来相当拗口。

文本的基本单位是字符向量，这意味着大部分字符串处理函数也能用于字符串向量，这与数学运算的向量化方式相同。

7.2.1 创建和打印字符串

如你所见，字符向量可用 `c` 函数创建。我们可以用单引号或双引号把字符串引用起来，只要引号之间匹配即可。不过，使用双引号更为标准：

```
c(
  "You should use double quotes most of the time",
  'Single quotes are better for including " inside the string'
)

## [1] "You should use double quotes most of the time"
## [2] "Single quotes are better for including \" inside the string"
```

`paste` 函数能将不同字符串组合在起来。在它传入的参数向量中，每个元素都能自我循环以达到最长的矢量长度，然后字符串就被拼接在一起，中间以空格分开。可以使用参数 `sep` 更改分隔符，或使用相关的 `paste0` 函数去掉分隔符。所有的字符串被组合后，可使用 `collapse` 参数把结果收缩成一个包含所有元素的字符串：

```
paste(c("red", "yellow"), "lorry")

## [1] "red lorry"    "yellow lorry"

paste(c("red", "yellow"), "lorry", sep = "-")

## [1] "red-lorry"    "yellow-lorry"

paste(c("red", "yellow"), "lorry", collapse = ", ")

## [1] "red lorry, yellow lorry"

paste0(c("red", "yellow"), "lorry")

## [1] "redlorry"    "yellowlorry"
```

`toString` 函数是 `paste` 的变种，它在打印向量时非常有用。它使用逗号和空格分隔每个元素，且可限制打印的数量。在下例中，`width =  40` 将输出限制为 40 个字符：

```
x <- (1:15) ^ 2
toString(x)

## [1] "1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225"

toString(x, width = 40)

## [1] "1, 4, 9, 16, 25, 36, 49, 64, 81, 100...."
```

cat 是一个低级函数，工作原理类似于 paste，不过格式功能更少。你将很少直接调用它。不过，你应该对它有所了解，因为它是大部分 print 函数的基础。cat 也可接受一个 file 参数[1]，将输出写入文件：

```
cat(c("red", "yellow"), "lorry")

## red yellow lorry
```

通常情况下，当字符串打印到控制台时，它们会以双引号括起来。如果对它们使用 noquote 函数，就可以去掉这些引号。有时，这会使文本更具可读性：

```
x <- c(
  "I", "saw", "a", "saw", "that", "could", "out",
  "saw", "any", "other", "saw", "I", "ever", "saw"
)
y <- noquote(x)
x

##  [1] "I"     "saw"   "a"     "saw"   "that"  "could" "out"   "saw"
##  [9] "any"   "other" "saw"   "I"     "ever"  "saw"

y

##  [1] I     saw   a     saw   that  could out   saw   any   other saw
## [12] I     ever  saw
```

7.2.2 格式化数字

有几个函数可用于数字的格式化。formatC 可让你使用 C 语言的格式化风格来指定使用固定型或科学型的格式、小数的位数以及输出的宽度。无论使用哪种选项，输入都应该是 numeric 类型（包括数组），且输出是 character 字符向量或数组：

```
pow <- 1:3
(powers_of_e <- exp(pow))

## [1]  2.718  7.389 20.086

formatC(powers_of_e)

## [1] "2.718" "7.389" "20.09"

formatC(powers_of_e, digits = 3)              # 指定三个数字

## [1] "2.72" "7.39" "20.1"

formatC(powers_of_e, digits = 3, width = 10)  # 前面加上一个空格

## [1] "      2.72" "      7.39" "      20.1"
```

注 1：再学究气点，它接受一个由 file 函数返回的文件的路径或连接。

```
formatC(powers_of_e, digits = 3, format = "e")  #科学格式

## [1] "2.718e+00" "7.389e+00" "2.009e+01"

formatC(powers_of_e, digits = 3, flag = "+")    #前面加上 +

## [1] "+2.72" "+7.39" "+20.1"
```

R 还提供了更通用的 C 风格的格式化函数 sprintf。与 sprintf 在其他任何语言中的工作方式一样：第一个参数包含字符串或数字变量的占位符，其他参数则将逐个代入这些占位符。不过请记住，R 中大部分的数值是浮点值而非整数。

sprint 的第一个参数指定了一个格式化字符串，其中包括其他值的占位符。例如：%s 代表另一个字符串，%f 和 %e 分别代表固定型格式和科学型格式的浮点数，%d 表示整数。其他参数的值将替换占位符。与 paste 函数类似，较短长度的输入将循环自身以匹配最长的输入：

```
sprintf("%s %d = %f", "Euler's constant to the power", pow, powers_of_e)

## [1] "Euler's constant to the power 1 = 2.718282"
## [2] "Euler's constant to the power 2 = 7.389056"
## [3] "Euler's constant to the power 3 = 20.085537"

sprintf("To three decimal places, e ^ %d = %.3f", pow, powers_of_e)

## [1] "To three decimal places, e ^ 1 = 2.718"
## [2] "To three decimal places, e ^ 2 = 7.389"
## [3] "To three decimal places, e ^ 3 = 20.086"

sprintf("In scientific notation, e ^ %d = %e", pow, powers_of_e)

## [1] "In scientific notation, e ^ 1 = 2.718282e+00"
## [2] "In scientific notation, e ^ 2 = 7.389056e+00"
## [3] "In scientific notation, e ^ 3 = 2.008554e+01"
```

其他格式化数字的方法有 format 和 prettyNum 这两个函数。format 提供的格式化字符串的语法稍有不同，与 formatC 的用法基本类似。而 prettyNum 则非常适合于格式化那些非常大或非常小的数字：

```
format(powers_of_e)

## [1] " 2.718" " 7.389" "20.086"

format(powers_of_e, digits = 3)                     #至少三个数字

## [1] " 2.72" " 7.39" "20.09"

format(powers_of_e, digits = 3, trim = TRUE)        #去掉多余的 0

## [1] "2.72" "7.39" "20.09"

format(powers_of_e, digits = 3, scientific = TRUE)
```

```
## [1] "2.72e+00" "7.39e+00" "2.01e+01"

prettyNum(
  c(1e10, 1e-20),
  big.mark = ",",
  small.mark = " ",
  preserve.width = "individual",
  scientific = FALSE
)

## [1] "10,000,000,000"           "0.00000 00000 00000 00001"
```

7.2.3 特殊字符

有一些特殊的字符可以被包含在字符串中。例如，我们可以通过 `\t` 插入一个制表符。在下例中，我们使用 `cat` 而非 `print`，因为 `print` 执行的额外的转换动作会把制表符 `\t` 转换成反斜杠和一个"t."。`cat` 的参数 `fill = TRUE` 使光标在一行结束后移动到下一行：

```
cat("foo\tbar", fill = TRUE)

## foo bar
```

将光标移动到下一行是通过打印换行符 `\n` 完成的（这在所有平台上都一样。在 R 中，不要使用 `\r` 或 `\r\n` 来打印换行符，因为 `\r` 会将光标移动到当前行的开始并覆盖你所写的内容）：

```
cat("foo\nbar", fill = TRUE)

## foo
## bar
```

在 R 的内部代码中，空字符 `\0` 用于终止字符串。然而，显式地把它们包含在字符串中是错误的（旧版本 R 会丢弃字符串中空字符之后的内容）：

```
cat("foo\0bar", fill = TRUE) # 这会抛出一个错误
```

打印反斜杠符时需要连续输入两个反斜杠符，以免被误认为特殊字符。在下例中，输入两个反斜杠，打印时只会看到一个：

```
cat("foo\\bar", fill = TRUE)

## foo\bar
```

如果我们需要在字符串中使用双引号，那么双引号符前必须加一个反斜杠来转义。同样地，如果要在字符串中使用单引号，则单引号需要被转义：

```
cat("foo\"bar", fill = TRUE)

## foo"bar
```

```
cat('foo\'bar', fill = TRUE)
```

```
## foo'bar
```

与之相反，如果在被双引号引用的字符串中使用单引号，或在被单引号引用的字符串中使用双引号，则并不需要对其进行转义：

```
cat("foo'bar", fill = TRUE)
```

```
## foo'bar
```

```
cat('foo"bar', fill = TRUE)
```

```
## foo"bar
```

通过打印报警符 `\a` 能让我们的电脑发出提示声（beep），不过 `alarm` 函数也能完成此功能且可读性更好。当想要程序在一个耗时很长的分析任务结束后主动通知你（你不在开放式的办公室），这个函数就能派上用场：

```
cat("\a")
alarm()
```

7.2.4　更改大小写

使用 `toupper` 和 `tolower` 函数能把字符串中的字符全部转换为大写或小写：

```
toupper("I'm Shouting")
```

```
## [1] "I'M SHOUTING"
```

```
tolower("I'm Whispering")
```

```
## [1] "i'm whispering"
```

7.2.5　截取字符串

有两个函数可用于从字符串中截取子串：`substring` 和 `substr`。在大多数情况下，你可以随便选一个使用。不过，如果你传入了不同长度的向量参数，它们的行为会略有不同。对 `substring` 来说，输出的长度与最长的输入一样；而对 `substr` 来说，输出的长度只与第一个输入的相等：

```
woodchuck <- c(
  "How much wood would a woodchuck chuck",
  "If a woodchuck could chuck wood?",
  "He would chuck, he would, as much as he could",
  "And chuck as much wood as a woodchuck would",
  "If a woodchuck could chuck wood."
)
```

```
substring(woodchuck, 1:6, 10)

## [1] "How much w" "f a woodc"  " would c"   " chuck "    " woodc"
## [6] "uch w"

substr(woodchuck, 1:6, 10)

## [1] "How much w" "f a woodc"  " would c"   " chuck "    " woodc"
```

7.2.6　分割字符串

paste 及其相关函数能把字符串组合在一起。strsplit 则正好相反，它在指定的某些点上分割字符串。我们可以把上例中的土拨鼠绕口字符串按空格分开。在下例中，fixed = TRUE 意味着 split 的参数是固定长度的字符串而非正则表达式：

```
strsplit(woodchuck, " ", fixed = TRUE)

## [[1]]
## [[1]]
## [1] "How"       "much"      "wood"      "would"     "a"         "woodchuck"
## [7] "chuck"
##
## [[2]]
## [1] "If"        "a"         "woodchuck" "could"     "chuck"     "wood?"
##
## [[3]]
##  [1] "He"     "would"  "chuck," "he"     "would," "as"     "much"
##  [8] "as"     "he"     "could"
##
## [[4]]
## [1] "And"       "chuck"     "as"        "much"      "wood"      "as"
## [7] "a"         "woodchuck" "would"
##
## [[5]]
## [1] "If"        "a"         "woodchuck" "could"     "chuck"     "wood."
```

请注意，strsplit 返回的是列表（而非字符向量或矩阵）。这是因为它的结果可能由不同长度的字符向量组成。当你只传入一个字符串时，这种情况很容易被忽视。请小心！

在我们的例子中，某些词最后的逗号有些烦人。最好的方法是在空格分割符后加一个可选的逗号，使用正则表达式就很容易搞定。? 意味着"前面的字符可选"：

```
strsplit(woodchuck, ",? ")

## [[1]]
## [1] "How"       "much"      "wood"      "would"     "a"         "woodchuck"
## [7] "chuck"
##
## [[2]]
## [1] "If"        "a"         "woodchuck" "could"     "chuck"     "wood?"
##
```

```
## [[3]]
##  [1] "He"    "would" "chuck" "he"    "would" "as"    "much"  "as"
##  [9] "he"    "could"
##
## [[4]]
## [1] "And"       "chuck"     "as"        "much"      "wood"      "as"
## [7] "a"         "woodchuck" "would"
##
## [[5]]
## [1] "If"        "a"         "woodchuck" "could"     "chuck"     "wood."
```

7.2.7 文件路径

R 有一个工作目录，默认为文件被读写的地方。我们可以使用 `getwd` 查看到它的位置，并使用 `setwd` 来改变它：

```
getwd()

## [1] "d:/workspace/LearningR"

setwd("c:/windows")
getwd()

## [1] "c:/windows"
```

请注意，每个路径的目录部分由正斜杠分隔，即使在 Windows 下也是这样。为了保持可移植性，在 R 中你可以始终对路径使用正斜杠。根据操作系统的不同，文件处理函数能够魔术般地把它们自动替换为反斜杠。

你也可以使用双反斜杠来表示 Windows 的路径，不过正斜杠仍为首选：

```
"c:\\windows"            # 记住使用两个斜杠
"\\\\myserver\\mydir"    # 对于 UNC 的命名法，需在开始使用四个斜杠
```

或者，你可以使用 `file.path` 来从各个目录中创建文件路径。它会自动地在目录名称之间插入正斜杠。它就像一个更加简单快速的为处理路径而定制的 `paste` 函数：

```
file.path("c:", "Program Files", "R", "R-devel")

## [1] "c:/Program Files/R/R-devel"

R.home()       # 同样也是 R 的安装目录

## [1] "C:/PROGRA~1/R/R-devel"
```

路径可以是绝对路径（从驱动器名称或网络共享处开始）或相对目录（相对于当前工作目录）。在后一种情况中，`.` 用于当前目录而 `..` 用于父目录。`~` 代表当前用户主目录。`path.expand` 能将相对路径转换为绝对路径：

```
path.expand(".")
```

```
## [1] "."
```

```
path.expand("..")
```

```
## [1] ".."
```

```
path.expand("~")
```

```
## [1] "C:\\Users\\richie\\Documents"
```

basename 只返回文件名，而不包括前面的目录位置。与之相反，dirname 只返回文件的目录：

```
file_name <- "C:/Program Files/R/R-devel/bin/x64/RGui.exe"
basename(file_name)
```

```
## [1] "RGui.exe"
```

```
dirname(file_name)
```

```
## [1] "C:/Program Files/R/R-devel/bin/x64"
```

7.3 因子

因子是一个用于存储类别变量的特殊的变量类型。它有时像字符串，有时又像整数。

7.3.1 创建因子

当你用一列文本数据创建数据框时，R 将文本默认为类别数据并进行转换。下例中的数据集包含了随机的 10 个成年人的身高数据：

```
(heights <- data.frame(
  height_cm = c(153, 181, 150, 172, 165, 149, 174, 169, 198, 163),
  gender = c(
    "female", "male", "female", "male", "male",
    "female", "female", "male", "male", "female"
 )
))
```

```
##    height_cm gender
## 1        153 female
## 2        181   male
## 3        150 female
## 4        172   male
## 5        165   male
## 6        149 female
## 7        174 female
## 8        169   male
## 9        198   male
## 10       163 female
```

通过检查 gender 一列的类，正如你所料，我们发现它不是一个字符向量，而是一个因子：

```
class(heights$gender)

## [1] "factor"
```

把这列打印出来可看到它更多的本质信息：

```
heights$gender

##  [1] female male   female male   male   female female male   male   female
## Levels: female male
```

因子中的每个值都是一个字符串，它们被限制为“female”、“male”或缺失值。如果我们把不同的字符串添加到 genders 中，此项约束则变得很明显：

```
heights$gender[1] <- "Female"    #注意是大写的 "F"

## Warning: invalid factor level, NA generated

heights$gender

##  [1] <NA>   male   female male   male   female female male   male   female
## Levels: female male
```

选项“female”和“male”被称为因子水平，它能用 levels 函数查询到：

```
levels(heights$gender)

## [1] "female" "male"
```

水平的级数（相当于因子 level 的 length）可由 nlevel 函数查询到：

```
nlevels(heights$gender)

## [1] 2
```

除了使用数据框在内部自动创建因子之外，也可以使用 factor 函数创建它。它的第一个参数（唯一的强制要求）是一个字符向量：

```
gender_char <- c(
  "female", "male", "female", "male", "male",
  "female", "female", "male", "male", "female"
)
(gender_fac <- factor(gender_char))

##  [1] female male   female male   male   female female male   male   female
## Levels: female male
```

7.3.2 更改因子水平

我们可以通过指定 `levels` 参数来更改因子被创建时水平的先后顺序：

```
factor(gender_char, levels = c("male", "female"))

##  [1] female male   female male   male   female female male   male   female
## Levels: male female
```

如果想在因子创建之后再改变因子水平的顺序，就再次使用 `factor` 函数，这时它的参数是当前的因子（而不是字符向量）：

```
factor(gender_char, levels = c("male", "female"))

##  [1] female male   female male   male   female female male   male   female
## Levels: male female
```

不能使用 `levels` 函数直接改变因子的水平值。因为这将重新为每一个水平打上标签，更改数据值，这通常不是我们想要的。

在下例中，直接设置因子的水平值会将数据中的 male 变成 female，female 变成 male，这并不是我们想要的：

```
levels(gender_fac) <- c("male", "female")
gender_fac

## [1] male    female male   female female male   male   female female male
## Levels: male female
```

`relevel` 函数是另一种改变因子水平顺序的方法。在这种情况下，你只需指定第一个水平值。正如你所料，此功能的使用场景还是相当少的（在将不同类别和引用类别相比较的回归模型中，它很好用）。大多数时候，使用 `factor` 函数来设置水平值会更方便：

```
relevel(gender_fac, "male")

##  [1] male   female male   female female male   male   female female male
## Levels: male female
```

7.3.3 去掉因子水平

在数据集清理的过程中，最终你可能需要去掉所有与因子水平对应的数据。考虑以下数据集，它记录了上班途中所使用的交通工具的次数：

```
getting_to_work <- data.frame(
  mode = c(
    "bike", "car", "bus", "car", "walk",
    "bike", "car", "bike", "car", "car"
```

```
  ),
    time_mins = c(25, 13, NA, 22, 65, 28, 15, 24, NA, 14)
  )
```

并非每次都有记录，所以我们的第一个任务是去掉那些 `time_mins` 是 `NA` 的行：

```
(getting_to_work <- subset(getting_to_work, !is.na(time_mins)))

##    mode time_mins
## 1  bike        25
## 2   car        13
## 4   car        22
## 5  walk        65
## 6  bike        28
## 7   car        15
## 8  bike        24
## 10  car        14
```

请看 `mode` 一列，其中只有三个不同值，而因子水平却是 four。我们可以使用 `unique` 函数（当然，也可以用 `levels` 函数）看清这一点：

```
unique(getting_to_work$mode)

## [1] bike car walk
## Levels: bike bus car walk
```

如果要删除未使用的水平因子，我们可以使用 `droplevels` 函数。它接受因子或是数据框作为参数。对后者来说，它将丢弃输入因子中所有未使用的水平。因为在我们的数据框里只有一个因子未使用，所以下例中的两行代码是等价的：

```
getting_to_work$mode <- droplevels(getting_to_work$mode)
getting_to_work <- droplevels(getting_to_work)
levels(getting_to_work$mode)

## [1] "bike" "car" "walk"
```

7.3.4　有序因子

有些因子的水平在语义上大于或小于其他水平。这在多选择题的调查问题中很常见。例如，调查问题“你有多快乐？”可能的回答包括：“depressed”“grumpy”“so-so”“cheery”和“ecstatic”[2]。结果是类别变量，所以我们可以创建一个拥有五个选项的因子。在此，使用 `sample` 函数来产生 10000 个随机回复：

```
happy_choices <- c("depressed", "grumpy", "so-so", "cheery", "ecstatic")
happy_values <- sample(
  happy_choices,
  10000,
```

注 2：有时这被称为李克特量（Likert scale）表。

```
  replace = TRUE
)
happy_fac <- factor(happy_values, happy_choices)
head(happy_fac)

## [1] grumpy    depressed cheery    ecstatic  grumpy    grumpy
## Levels: depressed grumpy so-so cheery ecstatic
```

在这种情况下，对调查者来说，五个选项其实是有顺序的："grumpy"（暴躁的）比"depressed"（沮丧）更快乐些，而"so-so"（马马虎虎）又比"grumpy"更快乐些等。这意味着最好把回复存储在一个按顺序排列的因子中。使用 `ordered` 函数（或通过给 `factor` 传入 `order = TRUE` 参数）可实现这个功能：

```
happy_ord <- ordered(happy_values, happy_choices)
head(happy_ord)

## [1] grumpy    depressed cheery    ecstatic  grumpy    grumpy
## Levels: depressed < grumpy < so-so < cheery < ecstatic
```

一个有序的因子还是因子，但一般的因子却不一定是有序的：

```
is.factor(happy_ord)

## [1] TRUE

is.ordered(happy_fac)

## [1] FALSE
```

在大多数情况下，你无需为有序因子的使用担心，而仅会在某些模型中看到差别。不过，在分析调查数据时，它们比较有用。

7.3.5 将连续变量转换为类别

一个汇总数值变量的方法是计算有多少个值落入不同的"组"（bins）中，`cut` 函数能将数值变量切成不同的块，然后返回一个因子。它通常使用 `table` 函数得到每组数字的总和。（`hist` 函数能画出直方图，它也能实现这个功能，就像我们将在后面看到的包含在 `plyr` 包中的 `count` 一样。）

在下例中，我们随机地生成 10 000 名工人的年龄数据（从 16 到 66，使用 Beta 分布），并将他们按每 10 年分组：

```
ages <- 16 + 50 * rbeta(10000, 2, 3)
grouped_ages <- cut(ages, seq.int(16, 66, 10))
head(grouped_ages)

## [1] (26,36] (16,26] (26,36] (26,36] (26,36] (46,56]
## Levels: (16,26] (26,36] (36,46] (46,56] (56,66]
```

```
table(grouped_ages)

## grouped_ages
## (16,26] (26,36] (36,46] (46,56] (56,66]
##    1844    3339    3017    1533     267
```

在本例中，大部分工人的年龄分布于 26 到 36 和 36 到 46 两个类别中（这正是使用 Beta 分布的结果）。

请注意，ages 是一个数字变量，而 grouped_ages 是一个因子：

```
class(ages)

## [1] "numeric"

class(grouped_ages)

## [1] "factor"
```

7.3.6 将类别变量转换为连续变量

与上面相反的情况是把因子转换成数值变量，这在数据清理中非常有用。如果你有一些脏数据，例如打错了的数字，在数据导入的过程中，R 会将它们解释为字符串，并将其转换为因子。在下例中，其中一个数字有两个小数点。诸如 read.table 的导入函数（第 12 章将深入探讨）将无法把这样的字符串解析成数字格式，而会默认把这一列转换成字符向量：

```
dirty <- data.frame(
  x = c("1.23", "4..56", "7.89")
)
```

要想把 x 列转换为数字，显而易见的解决方法是调用 as.numeric。可惜，它给出了错误的答案：

```
as.numeric(dirty$x)

## [1] 1 2 3
```

将 as.numeric 函数作用于因子上却看到了整数值，这说明因子使用了整数来存储数据。在一般情况下，可以从 levels(f)[as.integer(f)] 中重构出 f 因子。

为了正确地把因子转换为数字，可先把因子转换为字符向量，再转换为数值。第二个值是 NA，因为 4..56 并不是一个真正的数字：

```
as.numeric(as.character(dirty$x))

## Warning: NAs introduced by coercion

## [1] 1.23 NA 7.89
```

这不够高效，因为相同的值须被多次转换。正如 FAQ on R 中所指出的，更好的方法是将因子水平转换为数字，然后再像上面一样重建因子：

```
as.numeric(levels(dirty$x))[as.integer(dirty$x)]

## Warning: NAs introduced by coercion

## [1] 1.23   NA 7.89
```

因为这不够直观，所以如果你经常使用它，就把它包装成一个函数以方便调用：

```
factor_to_numeric <- function(f)
{
  as.numeric(levels(f))[as.integer(f)]
}
```

7.3.7　生成因子水平

为了平衡数据，使到在每个水平上的数据点的数目相等，可用 `gl` 函数来生成因子。它最简单的形式是：第一个整型参数为要生成的因子的水平数，第二个为每个水平需要重复的次数。通常你想为每个水平命名，这可以通过给 `label` 参数传入一个字符向量来实现。你还可以通过传入 `length` 参数以创建更复杂的水平排序，例如交替值（alternating value）：

```
gl(3, 2)

## [1] 1 1 2 2 3 3
## Levels: 1 2 3

gl(3, 2, labels = c("placebo", "drug A", "drug B"))

## [1] placebo placebo drug A  drug A  drug B  drug B
## Levels: placebo drug A drug B

gl(3, 1, 6, labels = c("placebo", "drug A", "drug B")) # 交替

## [1] placebo drug A  drug B  placebo drug A  drug B
## Levels: placebo drug A drug B
```

7.3.8　合并因子

如果我们有多个类别变量，有时把它们合并成一个单一的因子是有用的，其中每个水平由各个变量之间的交叉合并（使用 `interaction` 函数）组成：

```
treatment <- gl(3, 2, labels = c("placebo", "drug A", "drug B"))
gender <- gl(2, 1, 6, labels = c("female", "male"))
interaction(treatment, gender)

## [1] placebo.female placebo.male drug A.female drug A.male
## [5] drug B.female drug B.male
## 6 Levels: placebo.female drug A.female drug B.female ... drug B.male
```

7.4　小结

- 使用 `paste` 及其衍生函数能把字符串连接起来。
- 有很多函数可用于数字的格式化。
- 类别数据被存储在因子（或有序因子）中。
- 因子中每一个可能的类别被称为一个水平。
- 连续变量可以被 `cut` 函数转换成类别变量。

7.5　知识测试：问题

- 问题 7-1
  尽可能多地指出用于格式化数字的函数。

- 问题 7-2
  如何让你的电脑发出哔哔声？

- 问题 7-3
  类别变量的两种类型（type）的类（class）是什么？

- 问题 7-4
  如果你为一个因子添加一个值，但它不属于这个水平，会发生什么？

- 问题 7-5
  如何把数字变量转换成类别变量？

7.6　知识测试：练习

- 练习 7-1
  显示 `pi` 的 16 位有效数字。[5]

- 练习 7-2
  将以下字符串分割成单词，删除任何逗号或连字符：

```
x <- c(
  "Swan swam over the pond, Swim swan swim!",
  "Swan swam back again - Well swum swan!"
)
```

  [5]

- 练习 7-3
  在角色扮演游戏中，你的每个冒险家的人物属性计算公式为三个六面骰子得分的总和。

为了省下掷骰子的力气，你决定用 R 来生成分数。以下是用于生成它们的辅助函数：

```
#n 指定了要生成的得分，它必须是自然数
three_d6 <- function(n)
{
  random_numbers <- matrix(
    sample(6, 3 * n, replace = TRUE),
    nrow = 3
  )
  colSums(random_numbers)
}
```

得分多的将给予角色以奖励，得分少的则给予处罚，具体规则参照下表：

得　分	奖　励
3	–3
4,5	–2
6~ 8	–1
9~12	0
13~15	+1
16, 17	+2
18	+3

使用 `three_d6` 生成 1000 个人物属性得分。创建一个按奖励级别排列的得分表。[15]

第 8 章

流程控制和循环

与其他语言一样，R 在代码中会经常需要条件和重复逻辑控制存在。

如果你有其他编程语言的经验，就会对 R 中 `if` 和 `switch` 的功能非常熟悉，尽管 R 中的这些功能对你来说可能是新的。`ifelse` 函数是 R 的特殊函数之一，它是条件语句的向量化版本。

在本章中，我们将对它们以及其他三个你应该也非常熟悉的简单循环（`for`、`while` 和 `repeat`）逐一讲解。由于 R 的矢量化性质，以及其他更优雅的函数的存在，这些循语句在 R 中的应用不如预期的那样广泛。

8.1　本章目标

阅读本章后，你会了解以下内容：

- 如何使用分支语句；
- 如何使用循环语句重复执行代码。

8.2　流程控制

我们一般不满足于仅仅逐行地执行代码，而是希望更好地控制它们的执行流程。这就意味着只有在某些条件满足后你会才执行一些代码。

8.2.1　if和else

最简单的流程控制逻辑是使用 `if`。`if` 接受一个逻辑值（更准确地说是一个长度为 1 的逻辑

向量）作为参数，且当该值为 TRUE 时才会执行下一条语句：

```
if(TRUE) message("It was true!")

## It was true!

if(FALSE) message("It wasn't true!")
```

if 的条件中不允许缺失值，这样做会抛出一个错误：

```
if(NA) message("Who knows if it was true?")

## Error: missing value where TRUE/FALSE needed
```

如果你的条件中可能会出现缺失值，先用 is.na 来测试它：

```
if(is.na(NA)) message("The value is missing!")

## The value is missing!
```

当然，大部分时候你都不会直接传入 TRUE 或 FALSE 值，而是传递一个变量或表达式——因为如果知道该语句将被提前执行，就不需要 if 语句了。在下例中，runif(1) 将在 0 和 1 之间生成一个均匀分布的随机数。如果该值超过 0.5，则显示以下消息：

```
if(runif(1) > 0.5) message("This message appears with a 50% chance.")
```

如果你想有条件地执行多个语句，就把它们括在大括号中：

```
x <- 3
if(x > 2)
{
  y <- 2 * x
  z <- 3 * y
}
```

为了使代码更清晰，一些编程风格指南建议：即使条件执行语句只有一个，也要使用大括号。

与 if 对应的是 else 语句。如果 if 的条件值为 FALSE，则会执行 else 之后的代码：

```
if(FALSE)
{
  message("This won't execute...")
} else
{
  message("but this will.")
}

## but this will.
```

以下规则非常重要：else 必须与 if 语句的右大括号紧接在同一行。如果你把它挪到下一

行，将出现错误：

```
if(FALSE)
{
  message("This won't execute...")
}
else
{
  message("and you'll get an error before you reach this.")
}
```

你可以反复使用 `if` 和 `else` 来定义多个条件。请注意，`if` 和 `else` 仍然是两个独立的词——还有一个 `ifelse` 函数，它稍有不同，我们将马上看到：

```
(r <- round(rnorm(2), 1))

## [1] -0.1 -0.4

(x <- r[1] / r[2])

## [1] 0.25

if(is.nan(x))
{
  message("x is missing")
} else if(is.infinite(x))
{
  message("x is infinite")
} else if(x > 0)
{
  message("x is positive")
} else if(x < 0)
{
  message("x is negative")
} else
{
  message("x is zero")
}

## x is positive
```

与很多其他的语言不同，R 有一个小技巧能对你的代码重新排序来完成条件赋值。在下例中，`Re` 将返回复数实部（`Im` 将返回虚部）：

```
x <- sqrt(-1 + 0i)
(reality <- if(Re(x) == 0) "real" else "imaginary")

## [1] "real"
```

8.2.2　矢量化的`if`

标准的 `if` 语句需要一个逻辑值作为参数。如果给它传递一个长度超过 1 的逻辑向量（请不要这样做！），R 会警告你已给出多个选项，仅第一个将被使用：

```
if(c(TRUE, FALSE)) message("two choices")

## Warning: the condition has length > 1 and only the first element will be
## used

## two choices
```

由于 R 几乎都是矢量化的，你大概不会惊讶于矢量化的流程控制的存在，例如 `ifelse` 函数。`ifelse` 有三个参数：第一个是逻辑条件向量；第二个参数值在第一个向量为 `TRUE` 时被返回；第三个参数值在第一个向量为 `FALSE` 时被返回。在下例中，`rbinom` 使用二项分布生成随机数以模拟掷硬币过程：

```
ifelse(rbinom(10, 1, 0.5), "Head", "Tail")

## [1] "Head" "Head" "Head" "Tail" "Tail" "Head" "Head" "Head" "Tail" "Head"
```

`ifelse` 也可在第二个和第三个参数中接受向量。它们应与第一个向量的长度相等（如果不等，那么第二个和第三个参数中的元素将被重复或忽略，以使它们与第一个参数的长度相同）：

```
(yn <- rep.int(c(TRUE, FALSE), 6))

## [1] TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE
## [12] FALSE

ifelse(yn, 1:3, -1:-12)

##  [1]   1  -2   3  -4   2  -6   1  -8   3 -10   2 -12
```

如果条件参数中有缺失值，那么结果中的相应位置也是缺失值：

```
yn[c(3, 6, 9, 12)] <- NA
ifelse(yn, 1:3, -1:-12)

##  [1]   1  -2  NA  -4   2  NA   1  -8  NA -10   2  NA
```

8.2.3 多个分支

如果包含太多的 `else` 语句就会迅速降低代码的可读性。在这种情况下，可以用 `switch` 函数来美化它们。它的常见用法是：第一个参数为一个返回字符串的表达式，其后的参数为与第一个参数相匹配时的返回值。匹配参数必须与第一个参数完全一样（从 R 的 2.11.0 版本开始），且你可以使用大括号来执行多个表达式：

```
(greek <- switch(
  "gamma",
  alpha = 1,
  beta = sqrt(4),
  gamma =
  {
    a <- sin(pi / 3)
```

```
    4 * a ^ 2
  }
))
```

```
## [1] 3
```

如果找不到任何匹配的名字，那么 switch 将（隐式地）返回 NULL：

```
(greek <- switch(
  "delta",
  alpha = 1,
  beta = sqrt(4),
  gamma    =
  {
    a <- sin(pi / 3)
    4 * a ^ 2
  }
))
```

```
## NULL
```

对于这种情况，你可以在没有其他选择的情况下提供一个没有命名的参数：

```
(greek <- switch(
  "delta",
  alpha = 1,
  beta = sqrt(4),
  gamma =
  {
    a <- sin(pi / 3)
    4 * a ^ 2
  },
  4
))
```

```
## [1] 4
```

switch 的第一个参数也可以是一个整数。在这种情况下，其余的参数不需要名字——如果第一个参数结果为 1，那么将返回第二个参数的结果；如果第一个参数的结果为 2，则返回第三个参数的结果，以此类推：

```
switch(
  3,
  "first",
  "second",
  "third",
  "fourth"
)
```

```
## [1] "third"
```

你可能已经注意到了，在这种情况下没有默认参数。如果你要测试的整数非常大，这将相

当麻烦，因为你需要提供很多参数。这时，最好是把第一个参数转换为字符串，并且使用原来的第一种语法：

```
switch(
  as.character(2147483647),
  "2147483647" = "a big number",
  "another number"
)

## [1] "a big number"
```

8.3 循环

在 R 中有三种循环：`repeat`、`while` 和 `for`。虽然向量化意味着你可能并不像其他语言一样大量需要它们，但在需要重复执行代码时，它们还是很有用的。

8.3.1 重复循环

R 中最容易掌握的循环是 `repeat`。它所做的事情就是反复地执行代码，直到告诉它停为止。在其他语言中，一般使用 `do while` 或其他类似的方法完成。以下的例子[1]将反复执行，直到你按下 Escape 键、退出 R 或世界末日降临为止：

```
repeat
{
  message("Happy Groundhog Day!")
}
```

一般来说，我们还是希望在世界末日降临之前终止代码，因此需要一个 `break` 语句以跳出无限循环。在下例中，`sample` 函数将在每个循环迭代中返回一个操作：

```
repeat
{
   message("Happy Groundhog Day!")
  action <- sample(
    c(
        "Learn French",
        "Make an ice statue",
        "Rob a bank",
        "Win heart of Andie McDowell"
    ),
    1
  )
  message("action = ", action)
  if(action == "Win heart of Andie McDowell") break
}
```

注 1：如果这些例子对你来说没有意义，请观看电影 http://www.imdb.com/title/tt0107048。

```
## Happy Groundhog Day!

## action = Rob a bank

## Happy Groundhog Day!

## action = Rob a bank

## Happy Groundhog Day!

## action = Rob a bank

## Happy Groundhog Day!

## action = Win heart of Andie McDowell
```

有时候，我们想做的不是退出整个循环，而是跳过当前的迭代，开始 `next` 下一次迭代而已：

```
repeat
{
  message("Happy Groundhog Day!")
  action <- sample(
    c(
        "Learn French",
        "Make an ice statue",
        "Rob a bank",
        "Win heart of Andie McDowell"
    ),
    1
  )
  if(action == "Rob a bank")
  {
    message("Quietly skipping to the next iteration")
    next
  }
  message("action = ", action)
  if(action == "Win heart of Andie McDowell") break
}
```

```
## Happy Groundhog Day!

## action = Learn French

## Happy Groundhog Day!

## Quietly skipping to the next iteration

## Happy Groundhog Day!

## Quietly skipping to the next iteration

## Happy Groundhog Day!

## action = Make an ice statue

## Happy Groundhog Day!
```

```
## action = Make an ice statue

## Happy Groundhog Day!

## Quietly skipping to the next iteration

## Happy Groundhog Day!

## action = Win heart of Andie McDowell
```

8.3.2 while循环

while 循环就像是延迟了的 repeat 循环。它不是先执行代码再检查循环是否应该结束，而是先进行检查再（可能）执行代码。因为检查发生在开始时，所以循环体有可能不会被执行（与 repeat 循环不同）。在下例中，与以上 repeat 的例子类似，除了当“Win heart of Andie McDowell”被抽中了，剩下的 Groundhog Day 循环语句则可完全避免：

```
action <- sample(
  c(
    "Learn French",
    "Make an ice statue",
    "Rob a bank",
    "Win heart of Andie McDowell"
  ),
  1
)
while(action != "Win heart of Andie McDowell")
{
  message("Happy Groundhog Day!")
  action <- sample(
    c(
        "Learn French",
        "Make an ice statue",
        "Rob a bank",
        "Win heart of Andie McDowell"
    ),
    1
  )
  message("action = ", action)
}

## Happy Groundhog Day!

## action = Make an ice statue

## Happy Groundhog Day!

## action = Learn French

## Happy Groundhog Day!

## action = Make an ice statue

## Happy Groundhog Day!
```

```
## action = Learn French

## Happy Groundhog Day!

## action = Make an ice statue

## Happy Groundhog Day!

## action = Win heart of Andie McDowell
```

使用一些小技巧能把 repeat 循环转换为 while 循环，或把 while 循环转换为 loop 循环，但通常使用其中一种语法会更简洁。如果你知道循环体必须至少执行一次，请使用 repeat，否则使用 while。

8.3.3 for循环

第三种循环适用于已知代码所需执行的循环次数的情形。for 循环将接受一个迭代器变量和一个向量参数。在每个循环中，迭代器变量会从向量中取得一个值。最简单的情况下，该向量只包含整数：

```
for(i in 1:5) message("i = ", i)

## i = 1

## i = 2

## i = 3

## i = 4

## i = 5
```

如果你想执行多个表达式，与其他循环一样，须使用大括号把它们括起来：

```
for(i in 1:5)
{
  j <- i ^ 2
  message("j = ", j)
}

## j = 1

## j = 4

## j = 9

## j = 16

## j = 25
```

R 的 for 循环非常灵活，因为它们的输入并不限于整数或数字，还可以传入字符向量、逻辑向量或列表：

```
for(month in month.name)
{
  message("The month of ", month)
}

## The month of January

## The month of February

## The month of March

## The month of April

## The month of May

## The month of June

## The month of July

## The month of August

## The month of September

## The month of October

## The month of November

## The month of December

for(yn in c(TRUE, FALSE, NA))
{
  message("This statement is ", yn)
}
## This statement is TRUE
## This statement is FALSE
## This statement is NA

l <- list(
  pi,
  LETTERS[1:5],
  charToRaw("not as complicated as it looks"),
  list(
    TRUE
  )
)
for(i in l)
{
  print(i)
}

## [1] 3.142
## [1] "A" "B" "C" "D" "E"
##  [1] 6e 6f 74 20 61 73 20 63 6f 6d 70 6c 69 63 61 74 65 64 20 61 73 20 69
## [24] 74 20 6c 6f 6f 6b 73
## [[1]]
## [1] TRUE
```

因为 `for` 循环操作于向量中的每个元素，所以它提供了一种“伪向量化”。事实上，R 的向量化操作通常会在内部的 C 代码中使用某种形式的 `for` 循环。但要注意：R 的 `for` 循环几乎总是比其对应的向量化运行得要慢，而且往往是一到两个数量级的差别。这意味着你应尽可能地使用向量化[2]。

8.4 小结

- 使用 `if` 与 `else` 可以有条件地执行语句。
- `ifelse` 是它们所对应的向量化函数。
- R 有三种类型的循环：`repeat`、`while` 和 `for`。

8.5 知识测试：问题

- 问题 8-1
  如果为 `if` 传入条件 `NA` 会发生什么？

- 问题 8-2
  如果给 `ifelse` 传入条件 `NA` 会发生什么？

- 问题 8-3
  什么类型的变量可以作为 `switch` 函数的第一个参数传入？

- 问题 8-4
  如何中断一个 `repeat` 循环的执行？

- 问题 8-5
  如何跳转到循环的下一个迭代中？

8.6 知识测试：练习

- 练习 8-1
  在掷骰子游戏中，玩家（射手）准备掷出两个骰子，每个骰子都有六个面。如果掷出的总数为 2、3 或 12，则该射手失败。如果总数是 7 或 11，则对手（他）胜出。如果是其他任何得分，那么此得分将变成新的目标，它将被称为“点”。使用以下辅助函数为掷骰子生成得分：

  ```
  two_d6 <- function(n)
  {
    random_numbers <- matrix(
  ```

注 2：有公论的是，如果你把 R 代码编写得和 Fortran 一样，就没资格抱怨 R 运行得太慢了。

```
    sample(6, 2 * n, replace = TRUE),
    nrow = 2
  )
  colSums(random_numbers)
}
```

编写代码为掷骰子生成得分，并将以下值分配给 game_status 和 point 变量：

得　　分	游戏状态	点
2, 3, 11	FALSE	NA
7, 11	TRUE	NA
4, 5, 6, 8, 9, 10	NA	与得分相同

[10]

- 练习 8-2
  如果射手没有马上赢或马上输，那么他必须不断地滚动骰子，直到他赢得得分点值取胜，或得分为 7 而失败为止。编写代码检查游戏状态是否为 NA，如果是则反复生成掷骰子直到碰到点值为止（设置 game_status 为 TRUE）或得分为 7（设置 game_status 为 FALSE）。[15]

- 练习 8-3
  这是著名的 sea shells 绕口令：

```
sea_shells <- c(
  "She", "sells", "sea", "shells", "by", "the", "seashore",
  "The", "shells", "she", "sells", "are", "surely", "seashells",
  "So", "if", "she", "sells", "shells", "on", "the", "seashore",
  "I'm", "sure", "she", "sells", "seashore", "shells"
)
```

使用 nchar 函数来计算每个单词的字母数。现在，循环遍历所有可能的单词长度，找出所有与其长度相等的单词。例如，长度为 6 的单词应该有 shell 和 surely，它们都有六个字母。[10]

第 9 章

高级循环

R 的循环能力远远超出了上一章中所见到的三个标准循环。它能把你的函数应用于向量、列表或数组中的每个元素上，因此你可以写出一般矢量化所不能实现的“伪向量化代码”。其他循环则可以让你对每个数据块进行汇总统计。

9.1 本章目标

阅读本章后，你会了解以下内容：

- 如何把函数应用到列表或向量的每一个元素，或应用到矩阵的每一行或列上；
- 如何解决拆分 – 应用 – 合并（split-apply-combine）的问题；
- 如何使用 `plyr` 包。

9.2 `replication`

第 4 章介绍的 `rep` 函数能把输入的参数重复数次。另一个相关函数 `replicate` 则能调用表达式数次。大多数情况下它们基本相等，只有当使用随机数时才会出现不同。现在，假定生成均匀分布随机数的 `runif` 函数不是矢量化的，那么 `rep` 函数每次都将重复相同的随机数，而 `replicate` 每次的结果都不相同（由于历史的原因，其参数顺序竟然是从后到前的，这有点烦人）：

```
rep(runif(1), 5)

## [1] 0.04573 0.04573 0.04573 0.04573 0.04573
```

```
replicate(5, runif(1))

## [1] 0.5839 0.3689 0.1601 0.9176 0.5388
```

在更为复杂的例子中，replicate 会大显身手。例如，在蒙特卡罗（Monte Carlo）分析中——replicate 最主要的用途，你需要重复固定次数的分析过程且每次迭代都是相互独立的。

下一个例子将分析某人上下班时使用不同交通工具所花费的时间。这有些复杂，不过这是为了展示 replicate 的作用，它非常适合于这种场景。

time_for_commute 函数用 sample 随机挑选一种交通工具（小汽车、公交车或自行车），然后用 rnorm 或 rlnorm 找到一个正态分布或对数正态分布[1]的行程时间（具体参数取决于所选的交通工具）。

```
time_for_commute <- function()
{
  # 选择当时所用的交通工具
  mode_of_transport <- sample(
    c("car", "bus", "train", "bike"),
    size = 1,
    prob = c(0.1, 0.2, 0.3, 0.4)
  )
  # 根据交通工具，找到出行的时间
  time <- switch(
    mode_of_transport,
    car = rlnorm(1, log(30), 0.5),
    bus = rlnorm(1, log(40), 0.5),
    train = rnorm(1, 30, 10),
    bike = rnorm(1, 60, 5)
  )
  names(time) <- mode_of_transport
  time
}
```

switch 语句的存在使得这个函数很难被向量化。这意味着：为了找到上下班时间的分布，我们需要多次调用 time_for_commute 来生成每天的数据。replicate 使我们能即刻进行向量化：

```
replicate(5, time_for_commute())

## bike    car train   bus bike
## 66.22 35.98 27.30 39.40 53.81
```

9.3 遍历列表

现在，你已经注意到向量化在 R 中无处不在。事实上，你会很自然地选择编写向量化代码。因为它使代码看上去更精简，且与循环相比它的性能更好。不过，在某些情况下，保

注 1：对数正态分布偶尔会抛出非常大的数字，从而接近高峰期塞车。

持矢量化意味着控制代码的方式不太自然。此时，apply 系列的函数能更自然地让你进行“伪矢量化”[2]。

最简单且常用的成员函数是 lapply，它是“list apply”的缩写。lapply 的输入参数是某个函数，此函数将依次作用于列表中的每个元素上，并将结果返回到另一个列表中。回忆一下第 5 章介绍的质因数分解列表：

```
prime_factors <- list(
  two = 2,
  three = 3,
  four = c(2, 2),
  five = 5,
  six = c(2, 3),
  seven = 7,
  eight = c(2, 2, 2),
  nine = c(3, 3),
  ten = c(2, 5)
)
head(prime_factors)

## $two
## [1] 2
##
## $three
## [1] 3
##
## $four
## [1] 2 2
##
## $five
## [1] 5
##
## $six
## [1] 2 3
##
## $seven
## [1] 7
```

以向量化的方式在每个列表元素中搜索唯一值是很难做到的。我们可以写一个 for 循环来逐个地检查元素，但这种方法有点笨拙：

```
unique_primes <- vector("list", length(prime_factors))
for(i in seq_along(prime_factors))
{
  unique_primes[[i]] <- unique(prime_factors[[i]])
}
names(unique_primes) <- names(prime_factors)
unique_primes
```

注 2：由于向量化发生在 R 语言的级别，而非通过调用内部的 C 代码实现，所以它不能帮你得到更好的性能，仅使代码更可读。

```
## $two
## [1] 2
##
## $three
## [1] 3
##
## $four
## [1] 2
##
## $five
## [1] 5
##
## $six
## [1] 2 3
##
## $seven
## [1] 7
##
## $eight
## [1] 2
##
## $nine
## [1] 3
##
## $ten
## [1] 2 5
```

lapply 大大简化了这种操作，你无需再用那些陈腔滥调的代码来进行长度和名称检查：

```
lapply(prime_factors, unique)
```

```
## $two
## [1] 2
##
## $three
## [1] 3
##
## $four
## [1] 2
##
## $five
## [1] 5
##
## $six
## [1] 2 3
##
## $seven
## [1] 7
##
## $eight
## [1] 2
##
## $nine
```

```
## [1] 3
##
## $ten
## [1] 2 5
```

如果函数的每次返回值大小相同，且你知其大小为多少，那么你可以使用 lapply 的变种 vapply。vapply 的含义是：应用于（apply）列表而返回向量（vector）。和前面一样，它的输入参数是一个列表和函数，但 vapply 还需要第三个参数，即返回值的模板。它不直接返回列表，而是把结果简化为向量或数组：

```
vapply(prime_factors, length, numeric(1))
```

```
##   two three  four  five   six seven eight  nine   ten
##     1     1     2     1     2     1     3     2     2
```

如果输出不能匹配模板，那么 vapply 将抛出一个错误——vapply 不如 lapply 灵活，因为它输出的每个元素必须大小相同且必须事先就知道。

还有一种介于 lapply 和 vapply 之间的函数 sapply，其含义为：简化（simplfy）列表应用。与其他两个函数类似，sapply 的输入参数也是一个列表和函数。它不需要模板，但它会尽可能地把结果简化到一个合适的向量和数组中：

```
sapply(prime_factors, unique)   # 返回一个列表
```

```
## $two
## [1] 2
##
## $three
## [1] 3
##
## $four
## [1] 2
##
## $five
## [1] 5
##
## $six
## [1] 2 3
##
## $seven
## [1] 7
##
## $eight
## [1] 2
##
## $nine
## [1] 3
##
## $ten
## [1] 2 5
```

```
sapply(prime_factors, length) # 返回一个向量

##   two three  four  five   six seven eight  nine   ten
##     1     1     2     1     2     1     3     2     2

sapply(prime_factors, summary) # 返回一个数组

##         two three four five  six seven eight nine  ten
## Min.      2     3    2    5 2.00     7     2    3 2.00
## 1st Qu.   2     3    2    5 2.25     7     2    3 2.75
## Median    2     3    2    5 2.50     7     2    3 3.50
## Mean      2     3    2    5 2.50     7     2    3 3.50
## 3rd Qu.   2     3    2    5 2.75     7     2    3 4.25
## Max.      2     3    2    5 3.00     7     2    3 5.00
```

这非常适合于交互式的应用，因为你通常能自动地得到想要的形式。不过，如果你不太确定输入的是什么，那就要谨慎使用此函数。因为它的结果有时是一个列表，有时一个向量，会使你不知不觉地出错。之前的 `length` 的例子返回一个向量。现在，给它传入一个空列表时，看看会发生什么：

```
sapply(list(), length)

## list()
```

如果输入列表中的长度为零，无论函数传入了什么参数，`sapply` 总会返回一个列表。因此，如果你的数据是空的且你已知其返回值，使用 `vapply` 会更安全：

```
vapply(list(), length, numeric(1))

## numeric(0)
```

虽然这些函数主要和列表一起使用，但它们也可以接受向量为输入参数。在这种情况下，函数被依次应用到向量中的每个元素上。`source` 函数用于读取和访问 R 文件的内容（即可以用它来运行 R 脚本）。不幸地，它不是向量化的。因此，如果想运行某个目录下的所有 R 脚本，我们需要先把目录中的内容都转换到列表中再传给 `lapply`。

在下例中，`dir` 函数返回在指定目录中的文件名，默认为当前工作目录（回忆一下，你可以用 `getwd` 找到它）。参数 `pattern="\\.R$"` 的含义为：只返回以 `.R` 为后缀的文件名：

```
r_files <- dir(pattern = "\\.R$")
lapply(r_files, source)
```

你可能已经注意到，在所有的例子中，传到 `lapply`、`vapply` 和 `sapply` 的函数都只有一个参数。这些函数限制你只能传入一个向量化的参数（稍后会谈到如何规避此限制），但你可为其传入其他的标量参数。为此，只需把命名参数传递给 `lapply`（或 `sapply`、`vapply`），它们就会被传递到内部函数。例如，如果 `rep.int` 需要两个参数，而 `times` 参数只允许单个的（标量）数值，你可以输入：

```
complemented <- c(2, 3, 6, 18) # 参见 http://oeis.org/A000614
lapply(complemented, rep.int, times = 4)

## [[1]]
## [1] 2 2 2 2
##
## [[2]]
## [1] 3 3 3 3
##
## [[3]]
## [1] 6 6 6 6
##
## [[4]]
## [1] 18 18 18 18
```

如果向量参数不是第一个，那会如何？在这种情况下，我们需要自定义一个函数来封装那个真正想调用的函数。为此，你可以另起一行。但更常见的做法是把函数的定义包括在 `lapply` 的调用中：

```
rep4x <- function(x) rep.int(4, times = x)
lapply(complemented, rep4x)

## [[1]]
## [1] 4 4
##
## [[2]]
## [1] 4 4 4
##
## [[3]]
## [1] 4 4 4 4 4 4
##
## [[4]]
## [1] 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
```

通过把匿名函数传给 `lapply`，我们可以继续简化以上的代码。这是第五章所谈到的技巧：我们无需另起一行就能完成赋值操作，只要把函数传递给 `lapply` 即可，连名字都不需要：

```
lapply(complemented, function(x) rep.int(4, times = x))

## [[1]]
## [1] 4 4
##
## [[2]]
## [1] 4 4 4
##
## [[3]]
## [1] 4 4 4 4 4 4
##
## [[4]]
## [1] 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
```

在极个别的情况下，你可能需要循环遍历环境（而非列表）中每个变量。对此，你可以使

用专门的函数 eapply。当然，在最新版本的 R 中，你也可以使用 lapply：

```
env <- new.env()
env$molien <- c(1, 0, 1, 0, 1, 1, 2, 1, 3) # 参见 http://oeis.org/A008584
env$larry <- c("Really", "leery", "rarely", "Larry")
eapply(env, length)
```

```
## $molien
## [1] 9
##
## $larry
## [1] 4
```

```
lapply(env, length) # 一样的
```

```
## $molien
## [1] 9
##
## $larry
## [1] 4
```

rapply 是 lapply 函数的递归版本，它允许你循环遍历嵌套列表。这是个特殊的要求，且如果事先使用 unlist 将数据扁平化就会使代码变得更简单。

9.4 遍历数组

lapply 和它的小伙伴 vapply 与 sapply 都可用于矩阵和数组上，但它们的行为往往不是我们想要的。这三个函数把矩阵和数组看作向量，将目标函数作用于每个元素上（沿列往下移动）。而更为常见的是，当要把函数作用于一个数组时，我们希望能按行或列应用它们。下面的例子使用 matlab 包，提供了对手语言所具备的功能。

要运行下例，首先需要安装 matlab 程序包：

```
install.packages("matlab")
```

```
library(matlab)
```

```
## Attaching package: 'matlab'

## The following object is masked from 'package:stats':
##
## reshape

## The following object is masked from 'package:utils':
##
## find, fix

## The following object is masked from 'package:base':
##
## sum
```

> 当你装载 matlab 包时，它会覆盖一些在 base、stats 和 utils 包中的函数，把它们的行为重载成 MATLAB 中相应函数的行为。当使用完 matlab 包，你可能会想恢复以前的行为，这可以通过调用 detach("package:matlab") 来完成。

magic 函数将创建一个 f 方阵：n×n 的、从 1 排到 n^2 的数字矩阵，其行数和列数相等：

```
(magic4 <- magic(4))

##      [,1] [,2] [,3] [,4]
## [1,]   16    2    3   13
## [2,]    5   11   10    8
## [3,]    9    7    6   12
## [4,]    4   14   15    1
```

一个需要我们把函数应用到每行上的经典问题——计算行的总数，可以由在第五章中简要介绍的 rowSums 函数来实现：

```
rowSums(magic4)

## [1] 34 34 34 34
```

然而，如果要计算每行的其他统计值，该如何实现？要提供一个函数来实现所有可能的功能将是非常麻烦的[3]。apply 函数提供了类似的按行 / 列计算的等效函数，它以一个矩阵、维数和函数作为参数。维数为 1 代表把函数应用于每一行；2 代表把函数应用于每一列（或更大的数字代表更高维的数组）：

```
apply(magic4, 1, sum) # 与 rowSums 相同

## [1] 34 34 34 34

apply(magic4, 1, toString)

## [1] "16, 2, 3, 13" "5, 11, 10, 8" "9, 7, 6, 12" "4, 14, 15, 1"

apply(magic4, 2, toString)

## [1] "16, 5, 9, 4" "2, 11, 7, 14" "3, 10, 6, 15" "13, 8, 12, 1"
```

apply 也可用于数据框，尽管对于这种混合的数据类型来说不太常见（例如，如果其中有一列是字符类型的，很显然你不能在它上面计算总和或乘积）：

```
(baldwins <- data.frame(
  name             = c("Alec", "Daniel", "Billy", "Stephen"),
  date_of_birth    = c(
    "1958-Apr-03", "1960-Oct-05", "1963-Feb-21", "1966-May-12"
  ),
  n_spouses        = c(2, 3, 1, 1),
```

注 3：尽管 matrixStats 包试图这样做。

```
  n_children       = c(1, 5, 3, 2),
  stringsAsFactors = FALSE
))

##      name date_of_birth n_spouses n_children
## 1    Alec   1958-Apr-03         2          1
## 2  Daniel   1960-Oct-05         3          5
## 3   Billy   1963-Feb-21         1          3
## 4 Stephen   1966-May-12         1          2
```

```
apply(baldwins, 1, toString)

## [1] "Alec, 1958-Apr-03, 2, 1" "Daniel, 1960-Oct-05, 3, 5"
## [3] "Billy, 1963-Feb-21, 1, 3" "Stephen, 1966-May-12, 1, 2"
```

```
apply(baldwins, 2, toString)

##                                              name
##                       "Alec, Daniel, Billy, Stephen"
##                                     date_of_birth
## "1958-Apr-03, 1960-Oct-05, 1963-Feb-21, 1966-May-12"
##                                         n_spouses
##                                      "2, 3, 1, 1"
##                                        n_children
##                                      "1, 5, 3, 2"
```

当你把函数按列地应用于数据框上，apply 与 sapply 的行为相同（记住，数据框可被认为是非嵌套的列表，其中的元素都具有相同的长度）：

```
sapply(baldwins, toString)

##                                              name
##                       "Alec, Daniel, Billy, Stephen"
##                                     date_of_birth
## "1958-Apr-03, 1960-Oct-05, 1963-Feb-21, 1966-May-12"
##                                         n_spouses
##                                      "2, 3, 1, 1"
##                                        n_children
##                                      "1, 5, 3, 2"
```

当然，如果仅仅打印不同的形式数据集，你无法充分感受到它的趣味性。另一方面，将 sapply 与 range 结合使用能非常迅速地确定你的数据范围：

```
sapply(baldwins, range)

##      name      date_of_birth n_spouses n_children
## [1,] "Alec"    "1958-Apr-03" "1"       "1"
## [2,] "Stephen" "1966-May-12" "3"       "5"
```

9.5　多个输入的应用函数

lapply 的缺点之一是它的函数参数只能循环作用于单个向量参数。另一个缺点是：对于作

用于每个元素的函数，你不能访问该元素的名称。

mapply 是“多参数列表应用”（multiple argument list apply）的简称，它能让你传入尽可能多的向量作为参数，这解决了上面的第一个问题。常见的用法是传入一个列表，再传入另一个列表作为前者的名字，这就解决了第二个问题。有点烦人的是，为了容纳任意数目的向量参数，参数的顺序被改变了。对于 mapply，每一个传递的参数都是函数：

```
msg <- function(name, factors)
{
  ifelse(
    length(factors) == 1,
    paste(name, "is prime"),
    paste(name, "has factors", toString(factors))
  )
}
mapply(msg, names(prime_factors), prime_factors)

##                         two                       three
##              "two is prime"            "three is prime"
##                        four                        five
##     "four has factors 2, 2"             "five is prime"
##                         six                       seven
##      "six has factors 2, 3"            "seven is prime"
##                       eight                        nine
## "eight has factors 2, 2, 2"     "nine has factors 3, 3"
##                         ten
##      "ten has factors 2, 5"
```

在默认情况下，mapply 与 sapply 的表现相同，且会尽量地简化输出。通过传递参数 SIMPLIFY = FALSE 可以关闭此行为（使它表现得更像 lapply）。

即时向量化（Instant Vectorization）

Vectorize 是 mapply 的包装函数。通常它接受一个标量作为输入参数，并返回一个新的接受向量的函数。下一个函数不是向量化的，因为它使用了 switch，而这需要它的输入参数为标量：

```
baby_gender_report <- function(gender)
{
  switch(
    gender,
    male   = "It's a boy!",
    female = "It's a girl!",
    "Um..."
  )
}
```

如果把向量传入到函数中，则会抛出一个错误：

```
genders <- c("male", "female", "other")
baby_gender_report(genders)
```

从理论上说，完全可以重写一个函数，使它变成向量化的。但更简单的方法是使用 Vectorize 函数：

```
vectorized_baby_gender_report <- Vectorize(baby_gender_report)
vectorized_baby_gender_report(genders)

##          male        female         other
##  "It's a boy!" "It's a girl!"       "Um..."
```

9.6　拆分－应用－合并（Split-Apply-Combine）

在研究数据时一个很常见的问题是：如何对那些已被分成不同小组的变量进行统计计算。以下是经典的道路交通安全游戏 Frogger 的得分情况：

```
(frogger_scores <- data.frame(
  player = rep(c("Tom", "Dick", "Harry"), times = c(2, 5, 3)),
  score = round(rlnorm(10, 8), -1)
))

##    player score
## 1     Tom  2250
## 2     Tom  1510
## 3    Dick  1700
## 4    Dick   410
## 5    Dick  3720
## 6    Dick  1510
## 7    Dick  4500
## 8   Harry  2160
## 9   Harry  5070
## 10  Harry  2930
```

计算每个玩家的平均得分需要三个步骤。首先，我们按玩家来分开数据集：

```
(scores_by_player <- with(
  frogger_scores,
  split(score, player)
))

## $Dick
## [1] 1700  410 3720 1510 4500
##
## $Harry
## [1] 2160 5070 2930
##
## $Tom
## [1] 2250 1510
```

然后，我们将（mean）函数应用于每个元素：

```
(list_of_means_by_player <- lapply(scores_by_player, mean))

## $Dick
## [1] 2368
##
## $Harry
## [1] 3387
##
## $Tom
## [1] 1880
```

最后，我们把结果合并到单个向量中：

```
(mean_by_player <- unlist(list_of_means_by_player))

##  Dick Harry   Tom
##  2368  3387  1880
```

如果使用 vapply 或 sapply，最后两个步骤可简化成一步，但拆分－应用－合并是如此常用，所以我们必须简化它。方法就是使用 tapply 函数，它能一次执行所有的三个步骤，一气呵成：

```
with(frogger_scores, tapply(score, player, mean))

##  Dick Harry   Tom
##  2368  3387  1880
```

还有几个其他的 tapply 包装函数，例如 by 和 aggregate。它们的功能相同，但接口稍有不同。

SQL 的粉丝可能会注意到，拆分－应用－合并其实就是 GROUP BY 操作。

9.7 plyr包

*apply 函数家族都很强大，但三个缺点使得它们不是那么易用。首先，名字有点晦涩。在 lapply 中，“l”代表 list 是可理解的，但是，即便是使用 R 长达 9 年的我，也不知道 tapply 中的“t”代表什么。

其次，参数的使用不完全一致。大多数的函数都以数据对象为首个参数，函数为第二参数。但 mapply 的顺序却与此相反，而 tapply 还要加上一个函数作为它的第三个参数。数据参数有时是 X，有时是 object；而简化参数有时是 simplify，有时是 SIMPLIFY。

第三，输出的形式不太可控。如果要把结果作为数据框返回或丢弃它，都需要花一些心思才能做到。

这时，plyr 包就派上用场了。它包含一系列名为 **ply 的函数，其中的空格（星号）分别

代表输入和输出的形式。例如，llply 的输入参数是列表，它将函数应用于每个元素上，并返回一个列表，这使它成为 lapply 的一个替代函数：

```
library(plyr)
llply(prime_factors, unique)

## $two
## [1] 2
##
## $three
## [1] 3
##
## $four
## [1] 2
##
## $five
## [1] 5
##
## $six
## [1] 2 3
##
## $seven
## [1] 7
##
## $eight
## [1] 2
##
## $nine
## [1] 3
##
## $ten
## [1] 2 5
```

laply 以一个列表作为参数，并返回一个数组，这酷似 sapply。如果输入参数为空，它也会智能地返回一个空的逻辑向量（与返回空列表的 sapply 不一样）：

```
laply(prime_factors, length)

## [1] 1 1 2 1 2 1 3 2 2

laply(list(), length)

## logical(0)
```

raply 能取代 replicate（不是 rapply!），还有 rlply 和 rdply 函数能分别返回列表或数据框，还有一个 r_ply 函数能丢弃结果（在绘图时有用）：

```
raply(5, runif(1)) # 数组输出

## [1] 0.009415 0.226514 0.133015 0.698586 0.112846
```

```
rlply(5, runif(1)) # 列表给出

## [[1]]
## [1] 0.6646
##
## [[2]]
## [1] 0.2304
##
## [[3]]
## [1] 0.613
##
## [[4]]
## [1] 0.5532
##
## [[5]]
## [1] 0.3654

rdply(5, runif(1)) # 数据框输出

##   .n     V1
## 1 1 0.9068
## 2 2 0.0654
## 3 3 0.3788
## 4 4 0.5086
## 5 5 0.3502

r_ply(5, runif(1)) # 丢弃输出

## NULL
```

也许 plyr 包中最常用的函数是 ddply，它的输入和输出都是数据框，它可以替换 tapply 函数 。其优点是易于同时计算多个列。以下让我们为 Frogger 数据集添加一个 level 列，用来表示玩家在游戏中达到的级别：

```
frogger_scores$level <- floor(log(frogger_scores$score))
```

调用 ddply 的方法有好几种。这些方法的参数都包括一个数据框、要进行拆分的列的名称，以及要应用到每个元素上的函数。传递列时无需引号，但需要包含在 . 的调用之后。

对于函数，你既可以使用 colwise 告诉 ddply 调用函数应用于每一列（除了第二个参数以外），也可以使用 summarize 对指定的列进行操作：

```
ddply(
  frogger_scores,
  .(player),
  colwise(mean) # 除了 player 之外，对每个列调用 mean 函数
)

##   player score level
## 1  Dick 2368   7.200
## 2 Harry 3387   7.333
## 3   Tom 1880   7.000
```

```
ddply(
  frogger_scores,
  .(player),
  summarize,
  mean_score = mean(score), # 对 score 调用 mean
  max_level = max(level)    #... 然后求 level 的 max 值
)

##   player mean_score max_level
## 1  Dick        2368         8
## 2 Harry        3387         8
## 3   Tom        1880         7
```

使用 colwise 指定时会更快，但你必须对每一列重复同样的事情；而 summarize 更灵活，但你需要输入更多内容。

mapply 没有直接的替代函数，尽管 m*ply 函数允许对多个参数进行循环。同样，vapply 和 rapply 也没有替代函数 。

9.8 小结

- apply 函数家族能使循环语句更简洁。
- 当你要把数据进行分组时，拆分－应用－合并的问题是很常见的。
- 对于大多数 apply 函数来说，plyr 包提供了相应的替代函数且在语法上更为简洁。

9.9 知识测验：问题

- 问题 9-1

  列出尽可能多的 apply 函数家族中的成员函数。

- 问题 9-2

  lapply、vapply 和 sapply 之间的区别是什么？

- 问题 9-3

  你会如何遍历树状数据？

- 问题 9-4

  如果有一些身高的数据，如何按性别计算平均高度？

- 问题 9-5

  在 plyr 包中，**ply 中的星号意味着什么？

9.10 知识测试：练习

- 练习 9-1

  遍历名人 Wayans 家族中的儿童列表。Wayans 家庭中每一代人有多少个儿童？

```
wayans <- list(
  "Dwayne Kim" = list(),
  "Keenen Ivory" = list(
    "Jolie Ivory Imani",
    "Nala",
    "Keenen Ivory Jr",
    "Bella",
    "Daphne Ivory"
),
Damon = list(
  "Damon Jr",
  "Michael",
  "Cara Mia",
  "Kyla"
),
  Kim = list(),
  Shawn = list(
    "Laila",
    "Illia",
    "Marlon"
  ),
  Marlon = list(
    "Shawn Howell",
    "Arnai Zachary"
  ),
  Nadia = list(),
    Elvira = list(
    "Damien",
    "Chaunté"
  ),
  Diedre = list(
    "Craig",
    "Gregg",
    "Summer",
    "Justin",
    "Jamel"
  ),
  Vonnie = list()
)
```

  [5]

- 练习 9-2

  `state.x77` 是 R 提供的一个数据集，它包含了关于美国各州的人口、收入和其他信息。输入它的名字就可以看到它的值，和你自己创建数据集一样：

```
state.x77
```

(1) 使用你在第三章学到的方法检查数据集。

(2) 计算出每一列的平均值和标准差。

[10]

- 练习 9-3

  回顾本章前面部分介绍的 `time_for_commute` 函数。计算在 75% 位数的上下班时间，按不同的交通工具分组：

```
commute_times <- replicate(1000, time_for_commute())
commute_data <- data.frame(
  time = commute_times,
  mode = names(commute_times)
)
```

  [5]

第10章

包

R 中的软件包并不仅仅由 R 核心团队开发，更确切地说，是整个社区的努力使我们有成千上万的插件包可用于扩展。目前，大部分的包都安装在名为 CRAN（Comprehensive R Archive Network[1]）的在线资源库中 ，它由 R 核心团队维护。学会如何安装及使用这些插件包是 R 体验的一个重要部分。

在第 9 章中，我们介绍了 `plyr` 包。在本书接下来的部分，我们将学习更多的公共包。例如，用于日期和时间操作的 `lubridate` 包、用于导入 Excel 文件的 `xlsx`，用于操作数据框的 `reshape2`、用于绘图的 `ggplot2`，以及很多其他的包[2]。

10.1　本章目标

读完本章后，你会了解以下内容：

- 如何加载安装在机器上的包；
- 如何从本地文件或互联网安装新的包；
- 如何管理你电脑上的包。

10.2　加载包

你可以使用 `library` 函数来加载电脑上那些已经安装的包。但公认的更好的选择是使用

注 1：参照了 CPAN 的命名方式，Comprehensive Perl Archive Network。

注 2：你可以在 R-statistics blog 上（http://www.r-statistics.com/2013/06/top-100-r-packages-for-2013-jan-may）中找到一系列最流行的包。

`load_package`，它可以避免很多混淆。然而，因为 library 函数存在太久了，现在改变习惯为时已晚。就术语来说，包（package）是一系列 R 函数和数据集的集合；库（library）是你电脑上文件夹，而包就存储于文件夹内的文件中[3]。

如果你有一个标准版本的 R——也就是说，你还没有从 R 的源代码构建自定义的 R 版本——那么 `lattice` 包已经被默认安装了，但它不会自动加载。我们可以使用 `library` 函数加载它：

```
library(lattice)
```

现在，我们可以使用所有由 `lattice` 包所提供的函数。例如，图 10-1 显示了著名的 Immer's barley 数据集的点状图。

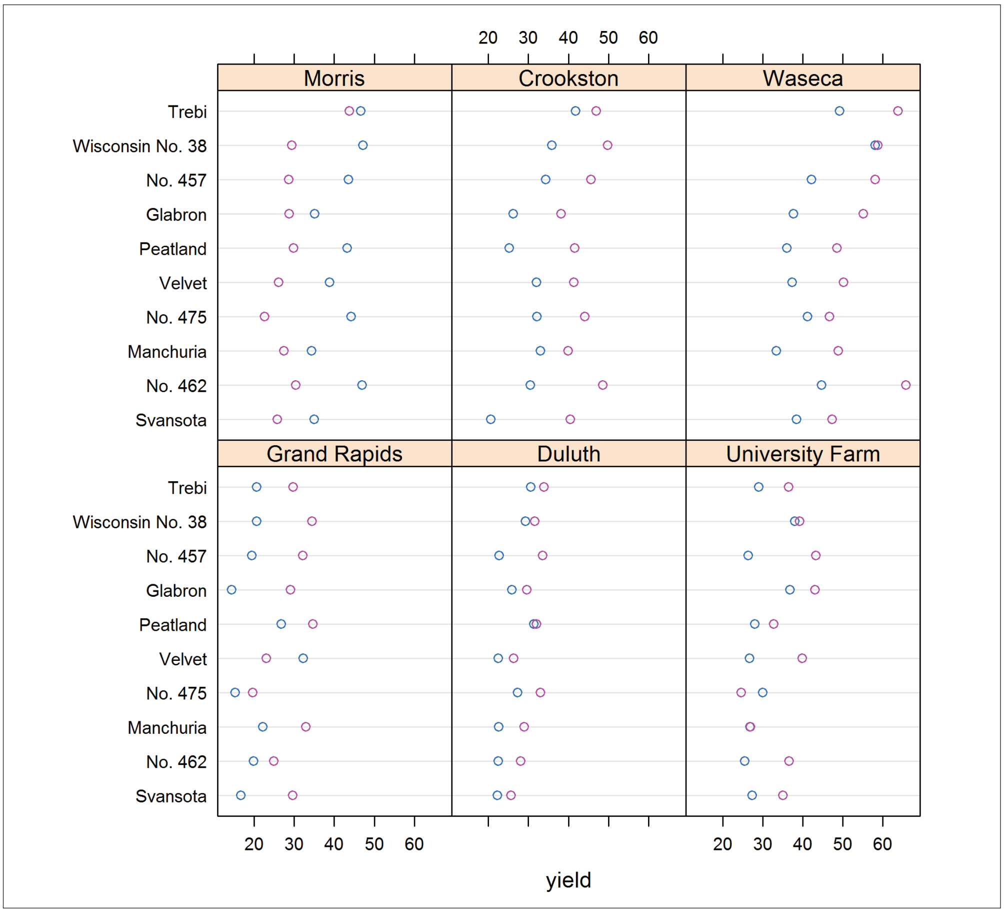


图 10-1：Immer's barley 数据集点状图

注 3：请注意，R-help 邮件列表中的一些人士认为把这两个术语混淆是罪大恶极的。

```
dotplot(
  variety ~ yield | site,
  data = barley,
  groups = year
)
```

lattice 包将在第 14 章详细讨论。

请注意，包的名称在传递给 library 库时并不需要被引号括起来。如果你想用编程方式把包的名字传递给 library，可以设置参数 character.only = TRUE。如果你有很多的包需要加载，这会有些用处：

```
pkgs <- c("lattice", "utils", "rpart")
for(pkg in pkgs)
{
  library(pkg, character.only = TRUE)
}
```

如果你使用 library 来加载一个未安装的包，它会抛出一个错误。如果你想用不同的方式来处理这种情况，可以试试 require 函数。和 library 一样，require 会加载一个包，不过它不会抛出一个错误而是通过判断包是否被成功加载而返回 TRUE 或 FALSE：

```
if(!require(apackagethatmightnotbeinstalled))
{
  warning("The package 'apackagethatmightnotbeinstalled' is not available.")
  # 或者尝试去下载它
  #...
}
```

10.2.1 搜索路径

你可以使用 search 函数查看所有已加载了的包：

```
search()

## [1] ".GlobalEnv"        "package:stats"     "package:graphics"
## [4] "package:grDevices" "package:utils"     "package:datasets"
## [7] "package:methods"   "Autoloads"         "package:base"
```

这个列表显示：R 是在哪里、以什么顺序搜索变量的。全局环境永远都是第一位的，其次是最近加载的包。最后两个值，一个是特殊的始终被称为 Autoloads 的环境，另一个是 base 包。当你在全局环境中定义了一个名为 var 的变量，在它从 stats 包中找到常用的方差函数之前，R 会首先在全局环境中发现它，因为全局环境始终在搜索列表的最前面。当你创建任何环境（参加第 6 章）时，它们也会出现在搜索路径中。

10.2.2 库和已安装的包

`installed.packages` 函数将返回一个数据框，它包含了 R 所知道的你电脑上所有包的信息。如果你使用 R 已经有一段时间了，包的数目很可能有几百个，所以最好还是远离控制台来查看结果：

```
View(installed.packages())
```

`installed.packages` 将为你提供电脑硬盘中每个版本的安装包的信息、它所依赖的其他包以及其他信息。在 `LibPath` 一列中提供了包的文件位置信息，这会告诉你它们的库在哪里。这时你可能想知道 R 如何决定哪些文件夹被认作是库。

> 以下的解释有些过于专业，你不必记住 R 是如何找到包这样的细节。这些信息在你选择升级 R，或当你加载包出现问题时能帮你节省一些时间，但在 R 的日常工作中不是必需的。

R 安装时就自带的包（`base`、`stats` 以及其他大概 30 个）都存储在你安装 R 的 library 子目录中。你可通过以下方法取得这个位置：

```
R.home("library") # 或者

## [1] "C:/PROGRA~1/R/R-devel/library"

.Library

## [1] "C:/PROGRA~1/R/R-devel/library"
```

在安装包时，你还得到一个只能由你访问的用户库（如果不希望你六岁大的小朋友更新家中电脑安装的包并破坏了代码的兼容性，这是非常有用的）。这个位置取决于操作系统。在 Windows 下，对于 R 的 x.y.z 版本，它在 Home 目录的 R/win-library/x.y 子目录中，Home 目录可以通过以下方式找到：

```
path.expand("~") #or

## [1] "C:\\Users\\richie\\Documents"

Sys.getenv("HOME")

## [1] "C:\\Users\\richie\\Documents"
```

在 Linux 下，文件夹同样位于 Home 目录之下的 R/R.version$platform-library/x.y 子文件夹中。`R.version$platform` 通常会返回一个类似于“i686-pc-linux-gnu”的字符串，你可以使用与 Windows 相同的方式找到它的 Home 录。在 Mac OS X 中，你可以在 Library/R/x.y/library 库中找到它。

与库位置的默认设置相关的一个问题是，当你升级 R 时需要重新安装所有的包。这是最安全的行为，因为不同版本的 R 常常需要使用不同版本的包。但在实际中，在开发机器上避免重新安装包往往要比版本问题更为重要[4]。为了方便，更好的方法是创建你自己的库，使之适用于 R 的所有版本。为此，最简单的方法是定义一个环境变量包 `R_LIBS`，它将包含你想要的库位置的路径[5]。虽然你可以通过在 R 中编程来定义环境变量，但它们只适用于 R，且仅用于会话的其余部分——而不是将它们定义在你的操作系统中。

你可以用 `.libPaths` 函数查看 R 所知道的所有字符向量：

```
.libPaths()

## [1] "D:/R/library"
## [2] "C:/Program Files/R/R-devel/library"
```

此向量的第一个值最为重要，因为这是包将被默认安装的位置。

10.3　安装包

R 出厂时被设置为访问 CRAN 的包库（提示选最近的镜像），如运行 Windows 则会访问 CRANextra。CRANextra 包含一些在 Windows 下构建时需特别注意的包，它们不能在通常的 CRAN 服务器上托管。要访问其他存储库，输入 `setRepositories()`，选择您想要的库。图 10-2 显示了它的可选项。

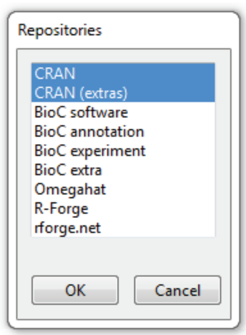


图 10-2：可用的包库列表

Bioconductor 包含了与基因组学和分子生物学的包，而 R-Forge 和 RForge.net 大多包含了开发中的版本，它们最终出现会在 CRAN。使用 `available.packages` 可以查看在库中的包的所有信息（请注意，因为那里的包有成千上万个，所以这需要好几秒钟运行）：

注 4：如果你正在把 R 程序部署为应用程序的一部分，那么情况就不同了。在这种情况下兼容性会胜出。
注 5：可以用分号分隔出多个位置，但这可能小题大做了。

```
View(available.packages())
```

除了这些库，还能从网上很多其他的储存库中找到 R 的包文件，如 GitHub、bitbucket 和 Google Code。正如下文所讨论的那样，从 GitHub 上取包特别容易。

许多 IDE 也提供了点选式的方法来安装包。 在 R 的 GUI 中，在 Package 菜单中提供了“Install Package(s) ...”的选项从资源库中安装包，以及使用“Install package(s) from local zip files...”来从你已下载的包中安装。图 10-3 显示 R 的 GUI 菜单。

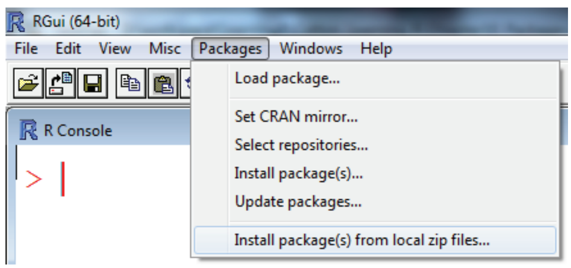


图 10-3：使用 R 的 GUI 来安装包

你也可以使用 `install.packages` 函数来安装包。不使用任何参数将弹出同样的 GUI 界面，就像你点击了“Install package(s) ...”菜单选项一样。通常你会想指定要安装的包的名字以及要访问的存储库的 URL 地址。在 CRAN 主站 http://cran.r-project.org/mirrors.html 上可以查到所有 CRAN 镜像的 RUL 列表。

此命令将（尝试）下载时间序列分析软件包 `xts`、`zoo` 和它们所有依赖的包，然后把它们安装到默认的库位置（`.libPaths` 返回的第一个值）：

```
install.packages(
  c("xts", "zoo"),
  repos = "http://www.stats.bris.ac.uk/R/"
)
```

如果想安装到不同的位置，把参数 `lib` 传递给 `install.packages`：

```
install.packages(
  c("xts", "zoo"),
  lib = "some/other/folder/to/install/to",
  repos = "http://www.stats.bris.ac.uk/R/"
)
```

很明显，你需要连上互联网才能使 R 下载包，而且你需要有把文件写入库目录的权限。在企业的内部网中，R 的互联网访问可能会受限。在 Windows 下，你可以让 R 使用 internet2.dll 来访问因特网，就像使用 IE 浏览器一样来绕过访问限制。要做到这一点，请输入：

```
setInternet2()
```

如果以上的方法都不可行，访问 http:// web/packages/available_packages_by_name.html 并手动下载你所需的包（以及所有依赖包），然后安装这些 tar.gz/tgz/zip 文件：

```
install.packages(
  "path/to/downloaded/file/xts_0.8-8.tar.gz",
  repos = NULL,       # repo 为 NULL 意味着包已经被下载
  type = "source"     # 这意味着现在就构建包
)

install.packages(
  "path/to/downloaded/file/xts_0.8-8.zip",
  repos = NULL,       # 还需要这个
  type = "win.binary" # 只是 Windows!
)
```

要直接从 GitHub 上安装包，你首先要安装 `devtools` 包：

```
install.packages("devtools")
```

`install_github` 函数接受 GitHub 库的名称，此库将包含你所需要的包（通常与包本身的名称一样），以及维护此库的作者名。例如，要得到开发版本的 reporting 包 `knitr`，输入：

```
library(devtools)
install_github("knitr", "yihui")
```

10.4 维护包

在包被安装后，你通常希望能及时更新它们以取得最新的版本。这可以通过 `update.packages` 完成。默认情况下，此函数会在更新每个包之前提示你。在一段时间后，这会变得很臃肿（因为安装几百个包的情况并不少见），所以建议设置 `ask = FALSE`：

```
update.packages(ask = FALSE)
```

有时候你可能想删除一个包。只需要简单地把包含此包的目录从你的文件系统中删除即可，或通过编程来做到这一点：

```
remove.packages("zoo")
```

10.5 小结

- 有成千上万的 R 包存在于在线资源库中。
- 可以使用 `install.packages` 来安装这些软件包；使用 `library` 或 `require` 来加载它们。
- 当加载包时，它们会被添加到 search 路径，由此 R 可以找到它们的变量。
- `installed.packages` 可以用于查看已安装的软件包；`update.packages` 用于更新它们；`remove.packages` 用于把它们从系统中删除。

10.6 知识测试：问题

- 问题 10-1
  请指出一些 R 包的资源库？

- 问题 10-2
  `library` 和 `require` 函数之间的区别是什么？

- 问题 10-3
  什么是包库（package library）？

- 问题 10-4
  如何才能在你的电脑上找到包库的位置？

- 问题 10-5
  如何才能使到 R 像 IE 浏览器一样上网？

10.7 知识测试：练习

- 练习 10-1
  使用 R 的 GUI 来安装 `Hmisc` 包。[10]

- 练习 10-2
  使用 `install.packages` 函数安装 `lubridate` 包。[10]

- 练习 10-3
  请统计出安装在你的电脑上的每个库下面的软件包的数量。[5]

第 11 章

日期和时间

日期和时间在数据分析中经常出现——尤其是对于时间序列分析来说。有一个坏消息：因为每月的天数、闰年和闰秒的存在 [1]，以及各个时区的天数都不相同，所以编写程序来处理它们相当麻烦。而好消息是：R 中有大量的函数可以用来处理时间和日期。尽管这些概念对于 R 编程来说是非常基础的，本书现在才讨论它们是因为其中一些最好的处理方法出现在插件包中。开始阅读本章时，你可能会对大量出现的代码感到不太自在。现在，我们将从 `lubridate` 包中开始探求，它能使你的日期－时间代码更具可读性。

11.1　本章目标

读完本章后，你会了解以下内容：

- 理解内置的日期类 `POSIXct`、`POSIXlt` 和 `date`；
- 如何将字符串转换成日期；
- 如何使用各种不同的格式显示日期；
- 如何指定和操作时区；
- 如何使用 `lubridate` 包。

11.2　日期和时间类

R 中自带有三个日期和时间类：`POSIXct`、`POSIXlt` 和 `Date`。

注 1：因为地球的旋转速度正在放缓，所以一天所需要的时间略长于 86 400 秒。这一点在你等待发工资时特别明显。从 1972 开始，我们通过添加闰秒来纠正这一点。输入 `.leap.seconds` 看它们什么时候发生。

11.2.1 POSIX日期和时间

POSIX 日期和时间是 R 的经典程序。它的实现精彩绝伦，考虑了各种晦涩难懂的技术问题。但它 Unix 系统的命名非常可怕，让一切看似比本身复杂得多。

R 中的两个标准的日期－时间类是 POSIXct 和 POSIXlt。（我说过它们的名字很吓人！）POSIX 是一套标准，它定义了那些需要遵守的 Unix 的规定，例如日期和时间该如何设定。ct 是“日历时间”（calendar time）的简称，POSIXct 类记录了以世界标准时（UTC）时区为准的从 1970 年开始计时的秒数计数[2]。POSIXlt 将日期存储为一个列表，其中包括秒、分钟、小时和月份等。POSIXct 最适用于存储和计算时间，而 POSIXlt 最适用于提取日期中的某个特定部分。

函数 Sys.time 将以 POSIXct 的形式返回当前的日期和时间：

```
(now_ct <- Sys.time())

## [1] "2013-07-17 22:47:01 BST"
```

now_ct 类有两个元素：一个是 POSIXct 变量，还有 POSIXct 继承自类 POSIXt：

```
class(now_ct)

## [1] "POSIXct" "POSIXt"
```

当日期被打印出来时，你只看到它格式化后的版本，因而不确定日期是如何被存储的。通过使用 unclass，我们发现它只是一个数字：

```
unclass(now_ct)

## [1] 1.374e+09
```

打印时，POSIXlt 日期看上去几乎都一样。然而，它们的底层存储机制是非常不同的：

```
(now_lt <- as.POSIXlt(now_ct))

## [1] "2013-07-17 22:47:01 BST"

class(now_lt)

## [1] "POSIXlt" "POSIXt"

unclass(now_lt)

## $sec
```

注 2：不要把 UTC 的缩写与其他通用时间标准相混合（如 UT0、UT1 等）。本质上，它和（民用的）格林尼治标准时间（GMT）一样，不同之处在于格林威治标准时间并非科学上的标准，而且英国政府也不能改变 UTC。

```
## [1] 1.19
##
## $min
## [1] 47
##
## $hour
## [1] 22
##
## $mday
## [1] 17
##
## $mon
## [1] 6
##
## $year
## [1] 113
##
## $wday
## [1] 3
##
## $yday
## [1] 197
##
## $isdst
## [1] 1
##
## attr(,"tzone")
## [1] ""    "GMT" "BST"
```

使用列表索引来单独访问 POSIXlt 日期的每个部分：

```
now_lt$sec
```

```
## [1] 1.19
```

```
now_lt[["min"]]
```

```
## [1] 47
```

11.2.2 Date类

在 R 的基本包中，第三种日期类命名得稍微好一点：它就是 Date 类。它存储了从 1970 年开始计算的天数[3]。Date 类最适用于当你不在乎一天中的某个时刻时。小数天也是可能的（例如可通过计算 Date 的平均值），但 POSIX 类更适合于这种情况：

```
(now_date <- as.Date(now_ct))
```

```
## [1] "2013-07-17"
```

注 3：历史数据研究人员可能要注意，这里的日期总是以公历计算，所以对于 1752 年之前的任何事情，都需要仔细检查。

```
class(now_date)

## [1] "Date"

unclass(now_date)

## [1] 15903
```

11.2.3 其他日期类

R 中还有其他许多处理处理日期和时间的类。如果需要选择日期－时间类，通常你应该坚持使用三个基本类（`POSIXct`、`POSIXlt` 和 `Date`）中的一个。但如果你正在使用其他人的代码，而这些代码使用了其他的日期－时间类，你就需要对其他类有所了解。

其他来自于各种插件包的日期时间类包括 `date`、`dates`、`chron`、`yearmon`、`yearqtr`、`timeDate`、`ti` 和 `jul`。

11.3 日期与字符串的相互转换

许多数据的文本文件格式并没有明确支持某种特定的日期类型。例如，在一个 CSV 文件中，每个值只是一个字符串。为了使用 R 中日期函数，你必须把日期字符串转换成某一个日期类型的变量。类似地，往 CSV 文件中写数据时，你也必须首先把日期转换成字符串。

11.3.1 解析日期

当我们从文本或电子表格文件读取日期时，它们通常被存储为一个字符向量或因子。为了把它们转换为日期，我们需要*解析*这些字符串。这可以使用另一种命名得同样糟糕的函数：`strptime`（string parse time 的简称），它将返回 `POSIXlt` 日期（还有 `as.POSIXct` 和 `as.POSIXlt` 函数。调用它们时，如果输入的是字符，那么它们只是 `strptime` 的封装函数罢了）。解析日期时，你必须把字符串与日期相对应的那些位告诉 `strptime`。日期的格式使用带有百分比符号和字母的字符串来指定。例如，当月的第几天被指定为 `%d`。把它们与其他固定字符相结合可以形成一个完整的规范，例如：冒号、破折号和日期中的斜杠号。时区的规范取决于你的操作系统。它可能很复杂，细节将在稍后讨论，但通常你希望把“UTC ”当成格林尼治时间或把“ ”当作当前时区（以你的操作系统的区域设置为主）。

在下例中，`%H` 是小时（24 小时制），`%M` 是分钟，`%S` 是秒，`%m` 是月数，`%d`（如前所述）是当月的第几天，还有 `%Y` 为四位数的年份。完整的组成列表在不同的系统上是不同的。详情请参见 `?strptime` 帮助页面：

```
moon_landings_str <- c(
  "20:17:40 20/07/1969",
  "06:54:35 19/11/1969",
```

```
  "09:18:11 05/02/1971",
  "22:16:29 30/07/1971",
  "02:23:35 21/04/1972",
  "19:54:57 11/12/1972"
)
(moon_landings_lt <- strptime(
  moon_landings_str,
  "%H:%M:%S %d/%m/%Y",
  tz = "UTC"
))

## [1] "1969-07-20 20:17:40 UTC" "1969-11-19 06:54:35 UTC"
## [3] "1971-02-05 09:18:11 UTC" "1971-07-30 22:16:29 UTC"
## [5] "1972-04-21 02:23:35 UTC" "1972-12-11 19:54:57 UTC"
```

如果字符串不匹配格式字符串中的格式，那么它就取 NA 值。例如，如果给出的是破折号而不是斜线，将使得解析失败：

```
strptime(
  moon_landings_str,
  "%H:%M:%S %d-%m-%Y",
  tz = "UTC"
)

## [1] NA NA NA NA NA NA
```

11.3.2 格式化日期

与解析相反的问题是如何把日期变量转换为字符串，即格式化。在这种情况下，我们将使用与指定格式字符串相同的系统，不过现在调用 strftime（字符串格式的时间）来反转解析操作。如果觉得 strftime 不好记，你可以使用 format 函数来轻松地完成日期的格式化，它与 strftime 的使用方式几乎完全相同。

在下例中，%I 表示小时（12 小时制），%p 是 AM/PM 指示，%A 是星期几的全名，而 %B 是月的全名。strftime 可使用 POSIXct 和 POSIXlt 的输入参数：

```
strftime(now_ct, "It's %I:%M%p on %A %d %B, %Y.")

## [1] "It's 10:47PM on Wednesday 17 July, 2013."
```

11.4 时区

从编程的角度来看，时区往往可怕而复杂。很多国家常有好几个时区，而且当一些（但不是全部）国家切换到夏令时需要改变边界。许多时区都有缩写名称，但它们又不是唯一的。例如，“EST”可以是美国、加拿大以及澳大利亚的“东部标准时间”。

在解析日期字符串时（使用 strptime），你可以指定一个时区；当你格式化它时（使用

strftime)，可以再次改变它。在解析过程中，如果不指定时区（默认为“”），R 会给日期以默认时区。这个值是 Sys.timezone 返回的，它会从你的操作系统区域设置猜到该值。你可以使用 Sys.getlocale("LC_TIME") 来查看操作系统的日期时间设定。

要避免这种时区的混乱，最简单的方法是随时记录，然后以 UTC 时区分析你的时间。如果你能做到这一点，恭喜！你很幸运。对于其他人，例如那些需要处理他人数据的人，最容易理解和方便的指定时区的方法是使用 Olson 形式，即“洲 / 市”或类似的方式：

```
strftime(now_ct, tz = "America/Los_Angeles")

## [1] "2013-07-17 14:47:01"

strftime(now_ct, tz = "Africa/Brazzaville")

## [1] "2013-07-17 22:47:01"

strftime(now_ct, tz = "Asia/Kolkata")

## [1] "2013-07-18 03:17:01"

strftime(now_ct, tz = "Australia/Adelaide")

## [1] "2013-07-18 07:17:01"
```

使用 file.path(R.home("share"), "zoneinfo", "zone.tab") 能查找出 R 中所有可能的 Olson 时区列表（这是名为 zone.tab 的文件位于 zoneinfo 文件夹中，此文件夹就在你安装 R 的共享目录内）。本章后文将介绍 lubridate 包，它将告诉你如何快速访问此文件。

另一个可靠的方法是给 UTC 手动添加一个偏移量，格式为 "UTC"+n 或 "UTC"-n。负的时间在 UTC 的东边，正的在西边。手动操作至少让你很清楚时间是如何被改变的，但同时你必须手动修正夏令时，因此这种方法须谨慎使用。最近版本的 R 会警告时区是未知的，但仍会正确执行此偏移操作：

```
strftime(now_ct, tz = "UTC-5")

## Warning: unknown timezone 'UTC-5'

## [1] "2013-07-18 02:47:01"

strftime(now_ct, tz = "GMT-5")     #同样的结果

## Warning: unknown timezone 'GMT-5'

## [1] "2013-07-18 02:47:01"

strftime(now_ct, tz = "-5")         # 如果操作系统支持，结果也是一样

## Warning: unknown timezone '-5'

## [1] "2013-07-18 02:47:01"
```

```
strftime(now_ct, tz = "UTC+2:30")
## Warning: unknown timezone 'UTC+2:30'
## [1] "2013-07-17 19:17:01"
```

指定时区的第三个方法是使用缩写，可以是三个字母或三个字母加一个数字再加三个字母的方式。这种方法不到最后不要使用，原因有三个：首先，缩写难以阅读，更容易出错；其次，正如前面提到的，它们不是唯一的，所以给出的时区可能不是你所要的；最后，不同的操作系统支持不同的缩写集。尤其对于 Windows 操作系统来说，它对时区缩写的处理比较别扭：

```
strftime(now_ct, tz = "EST")       # 加拿大东部时间
## [1] "2013-07-17 16:47:01"
strftime(now_ct, tz = "PST8PDT")   # 太平洋标准夏令时时间
## [1] "2013-07-17 14:47:01"
```

最后的关于时区的警告是：`strftime` 将忽略 `POSIXlt` 类型的时区变更。最好在打印之前就显式把日期转换成 `POSIXct` 类型：

```
strftime(now_ct, tz = "Asia/Tokyo")
## [1] "2013-07-18 06:47:01"
strftime(now_lt, tz = "Asia/Tokyo")           # 时区没有变化！
## [1] "2013-07-17 22:47:01"
strftime(as.POSIXct(now_lt), tz = "Asia/Tokyo")
## [1] "2013-07-18 06:47:01"
```

还有一个最后警告（真的是最后一个啦！）：如果你调用带有 `POSIXlt` 参数的连接函数 `c`，它会把时区更改为当地时区。相反，如果把 `c` 函数作用于 `POSIXct` 参数上，将彻底去除其时区属性。(大多数其他的函数都将假定日期是本地时区的，但要小心！)

11.5 日期和时间的算术运算

R 支持三个日期与时间基类的算术运算。将数字与 `POSIX` 日期相加，会以秒为单位增加时间。将数字与 `Date` 相加会以天数为单位：

```
now_ct + 86400      # 明天。我在想世界会变成怎么样!
## [1] "2013-07-18 22:47:01 BST"
```

```
now_lt + 86400     # 与 POSIXlt的行为一样
## [1] "2013-07-18 22:47:01 BST"

now_date + 1       # 日期的算术运算以天为单位

## [1] "2013-07-18"
```

把两个日期加起来其实没有多大意义，而且会抛出一个错误。但减法操作是支持的，这会计算两个日期之间的差值。这种行为对于所有三种日期的类型都一样。请注意，在下例中，如果你不指定格式的话，as.Date 会自动解析 %Y-%m-%d 或 %Y/%m/%d 形式的日期：

```
the_start_of_time <-   # 按 POSIX
  as.Date("1970-01-01")
the_end_of_time <-     # 按玛雅人的阴谋论
  as.Date("2012-12-21")
(all_time <- the_end_of_time - the_start_of_time)

## Time difference of 15695 days
```

使用我们已熟稔的（希望如此）class 和 unclass 的组合来查看时间存储之间的差别：

```
class(all_time)

## [1] "difftime"

unclass(all_time)

## [1] 15695
## attr(,"units")
## [1] "days"
```

使用 difftime 函数来计算出日期和时间之间的差值，它以数字的形式存储并以天为单位。由于时间之间的差别，天数被自动选择为“最敏感的”单位。差别短于一天的时间以小时、分钟或秒来表示。你可以使用 difftime 函数来更好地控制它的单位：

```
difftime(the_end_of_time, the_start_of_time, units = "secs")

## Time difference of 1.356e+09 secs

difftime(the_end_of_time, the_start_of_time, units = "weeks")

## Time difference of 2242 weeks
```

生成序列的 seq 函数也适用于日期。这在创建人工生成的日期（artificial date）的测试数据集时尤其有用。在 by 参数中，单位的选择对于 POSIX 和 date 类型是有所不同的。参考 ?seq.POSIXt 和 ?seq.Date 帮助页面以了解每种状况下应做何选择：

```
seq(the_start_of_time, the_end_of_time, by = "1 year")

##  [1] "1970-01-01" "1971-01-01" "1972-01-01" "1973-01-01" "1974-01-01"
##  [6] "1975-01-01" "1976-01-01" "1977-01-01" "1978-01-01" "1979-01-01"
```

```
## [11] "1980-01-01" "1981-01-01" "1982-01-01" "1983-01-01" "1984-01-01"
## [16] "1985-01-01" "1986-01-01" "1987-01-01" "1988-01-01" "1989-01-01"
## [21] "1990-01-01" "1991-01-01" "1992-01-01" "1993-01-01" "1994-01-01"
## [26] "1995-01-01" "1996-01-01" "1997-01-01" "1998-01-01" "1999-01-01"
## [31] "2000-01-01" "2001-01-01" "2002-01-01" "2003-01-01" "2004-01-01"
## [36] "2005-01-01" "2006-01-01" "2007-01-01" "2008-01-01" "2009-01-01"
## [41] "2010-01-01" "2011-01-01" "2012-01-01"

seq(the_start_of_time, the_end_of_time, by = "500 days") # 夏天

##  [1] "1970-01-01" "1971-05-16" "1972-09-27" "1974-02-09" "1975-06-24"
##  [6] "1976-11-05" "1978-03-20" "1979-08-02" "1980-12-14" "1982-04-28"
## [11] "1983-09-10" "1985-01-22" "1986-06-06" "1987-10-19" "1989-03-02"
## [16] "1990-07-15" "1991-11-27" "1993-04-10" "1994-08-23" "1996-01-05"
## [21] "1997-05-19" "1998-10-01" "2000-02-13" "2001-06-27" "2002-11-09"
## [26] "2004-03-23" "2005-08-05" "2006-12-18" "2008-05-01" "2009-09-13"
## [31] "2011-01-26" "2012-06-09"
```

许多其他基本的函数也允许操作日期。对它们可以使用 repeat、round 以及 cut 函数。也可以使用 mean 和 summary 来计算均值和汇总统计值。使用 methods(class = "POSIXt") 以及 methods(class = "Date") 能查看很多相关的函数，尽管其中一些函数在处理日期时，没有特定的日期方法。

11.6 lubridate

如果你对日期感到有些灰心，正想跳过本章的其余部分，不要害怕，因为救星来了！lubridate，正如其名，它为日期处理的过程增加了急需的润滑剂。它并没有为 R 添加许多新功能，但它使代码更具可读性，并免去你很多不必要的烦恼。

为了取代 strptime，lubridate 有多种使用了预设格式的解析函数。ymd 接受年、月、日的日期形式。规范上有一定的灵活性：一些基本的分隔符都能使用，如连字符、正反斜杠、冒号和空格[4]；月份可用数字、完整的英文名或缩写名称来标注；也可以使用星期几，但不是强制的。它真正的优点在于，相同向量中的不同元素可用不同的格式（只要年月日的先后顺序不变）：

```
library(lubridate)

## Attaching package: 'lubridate'

## The following object is masked from 'package:chron':
##
## days, hours, minutes, seconds, years

john_harrison_birth_date <- c( # 它发明了潜艇中的航海天文钟
   "1693-03 24",
   "1693/03\\24",
   "Tuesday+1693.03*24"
```

注 4：事实上，大部分的标点符号都是允许的。

```
)
ymd(john_harrison_birth_date) # 所有都一样

## [1] "1693-03-24 UTC" "1693-03-24 UTC" "1693-03-24 UTC"
```

非常重要的是，请记住 ymd 要以正确的顺序获取日期。如果你的日期数据的形式有所不同，可以使用 lubridate 提供的其他函数（ydm、mdy、myd、dmy 和 dym）。这些函数都有相关的函数用于指定特定的时间格式，如 ymd_h、ymd_hm 和 ymd_hms，以及另五个以其他顺序出现的日期函数。如果你的日期不在以上任何一种格式中，则使用更低级的函数 parse_date_time 来实现。

lubridate 中的所有解析函数都会返回 POSIXct 日期，默认都使用 UTC 时区。警告：这些行为与 R 中的基本函数 strptime 不同！（尽管通常使用起来会更方便。）在 lubridate 的术语中，这些个别日期是“瞬间”（instant）。

lubridate 提供了 stamp 函数来格式化日期，使你以更可读的方式指定格式。当指定一个日期，它会返回一个用于日期格式化的函数：

```
date_format_function <-
  stamp("A moon landing occurred on Monday 01 January 1900 at 18:00:00.")

## Multiple formats matched: "A moon landing occurred on %A %m January %d%y
## at %H:%M:%OS"(1), "A moon landing occurred on %A %m January %Y at
## %d:%H:%M."(1), "A moon landing occurred on %A %d %B %Y at %H:%M:%S."(1)

## Using: "A moon landing occurred on %A %d %B %Y at %H:%M:%S."

date_format_function(moon_landings_lt)

## [1] "A moon landing occurred on Sunday 20 July 1969 at 20:17:40."
```

lubridate 有三种不同类型的变量可用于时间范围的处理。“持续时间”（Duration）指定的时间跨度为秒的倍数，所以一天的总时间是 86 400 秒（60×60×24），一年的总时间是 3 156 000 秒（86 400×365）。对于间隔均匀的时间来说，指定日期范围会很容易，但闰年和夏令时把问题复杂化了。在下例中，请注意每个闰年会使日期向后倒退一天。today 将给出今天的日期：

```
(duration_one_to_ten_years <- dyears(1:10))

## [1] "31536000s (~365 days)" "63072000s (~730 days)"
## [3] "94608000s (~1095 days)" "126144000s (~1460 days)"
## [5] "157680000s (~1825 days)" "189216000s (~2190 days)"
## [7] "220752000s (~2555 days)" "252288000s (~2920 days)"
## [9] "283824000s (~3285 days)" "315360000s (~3650 days)"

today() + duration_one_to_ten_years

## [1] "2014-07-17" "2015-07-17" "2016-07-16" "2017-07-16" "2018-07-16"
## [6] "2019-07-16" "2020-07-15" "2021-07-15" "2022-07-15" "2023-07-15"
```

其他用于创建持续时间的函数为 `dseconds`、`dminutes` 等，对于混合组分的规范（mixed-component specification）则使用 `new_duration`。

“周期”（period）根据时钟上的时间来指定时间跨度。这意味着，在把它们添加到一个瞬间之前，它们确切的时间跨度是看不出来的。例如，一年的周期可以是 365 或 366 天，这取决于它是否是闰年。在下例中，请注意日期在闰年中保持不变：

```
(period_one_to_ten_years <- years(1:10))

##  [1] "1y 0m 0d 0H 0M 0S" "2y 0m 0d 0H 0M 0S" "3y 0m 0d 0H 0M 0S"
##  [4] "4y 0m 0d 0H 0M 0S" "5y 0m 0d 0H 0M 0S" "6y 0m 0d 0H 0M 0S"
##  [7] "7y 0m 0d 0H 0M 0S" "8y 0m 0d 0H 0M 0S" "9y 0m 0d 0H 0M 0S"
## [10] "10y 0m 0d 0H 0M 0S"

today() + period_one_to_ten_years

## [1] "2014-07-17" "2015-07-17" "2016-07-17" "2017-07-17" "2018-07-17"
## [6] "2019-07-17" "2020-07-17" "2021-07-17" "2022-07-17" "2023-07-17"
```

除了 `years`，还可以使用 `seconds`、`minutes` 以及 `new_period`（对混合组分规范）等来创建周期。

“间隔”（interval）定义为某段具有开始和结束的时间。它们本身用处不大，最常用于指定持续时间和周期，当你已知开始和结束日期（而非持续的时间）时 。它们也可用于持续时间和周期之间的转换。例如，如果要将一年的持续时间直接转换为周期只能大约估计，因为一年可能是 365 或 366 天（可能再加上一些闰秒，如果是夏令时变化则要加减一、两个小时）：

```
a_year <- dyears(1) # 刚好是 60*60*24*365 秒
as.period(a_year)   # 只是一个估计

## estimate only: convert durations to intervals for accuracy

## [1] "1y 0m 0d 0H 0M 0S"
```

当起始（或结束）时间的日期已知时，我们可以使用 `interval` 和一个媒介把持续时间转换为周期，例如：

```
start_date <- ymd("2016-02-28")
(interval_over_leap_year <- new_interval(
  start_date,
  start_date + a_year
))

## [1] 2016-02-28 UTC--2017-02-27 UTC

as.period(interval_over_leap_year)

## [1] "11m 30d 0H 0M 0S"
```

间隔也有一些便于使用的操作符，即用于定义间隔的 %--%，以及用于检查日期是否包含在间隔之内的 %within%：

```
ymd("2016-02-28") %--% ymd("2016-03-01") # 另一种指定间隔的方式

## [1] 2016-02-28 UTC--2016-03-01 UTC

ymd("2016-02-29") %within% interval_over_leap_year

## [1] TRUE
```

对于时区的处理，with_tz 允许你改变日期的时区而无须把它打印出来（这与 strftime 不同）。它还能够正确地处理 POSIXlt 日期（这也与 strftime 不同）：

```
with_tz(now_lt, tz = "America/Los_Angeles")

## [1] "2013-07-17 14:47:01 PDT"

with_tz(now_lt, tz = "Africa/Brazzaville")

## [1] "2013-07-17 22:47:01 WAT"

with_tz(now_lt, tz = "Asia/Kolkata")

## [1] "2013-07-18 03:17:01 IST"

with_tz(now_lt, tz = "Australia/Adelaide")

## [1] "2013-07-18 07:17:01 CST"
```

force_tz 是 with_tz 的一个变种，用于更新不正确的时区。

olson_time_zones 将以字母或经度顺序返回 R 所知道的所有 Olson 风格的时区名字列表：

```
head(olson_time_zones())

## [1] "Africa/Abidjan" "Africa/Accra" "Africa/Addis_Ababa"
## [4] "Africa/Algiers" "Africa/Asmara" "Africa/Bamako"

head(olson_time_zones("longitude"))

## [1] "Pacific/Midway" "America/Adak" "Pacific/Chatham"
## [4] "Pacific/Wallis" "Pacific/Tongatapu" "Pacific/Enderbury"
```

还一些其他的实用函数可用于日期的算术运算，尤其是 floor_date 和 ceiling_date：

```
floor_date(today(), "year")

## [1] "2013-01-01"

ceiling_date(today(), "year")

## [1] "2014-01-01"
```

11.7 小结

- 有三个可用于存储日期和时间的内置类：`POSIXct`、`POSIXlt` 和 `Date`。
- 使用 `strptime` 函数解析字符，把它转换为日期。
- 使用 `strftime` 把日期格式化为字符串。
- 使用 Olson 名称或 UTC 的偏移量、或（有时）用三个字母的缩写指定时区。
- `lubridate` 包让处理时间变得稍容易些。

11.8 知识测试：问题

- 问题 11-1
  你会使用三个内建类的哪一个来存储数据框中的日期与时间？

- 问题 11-2
  `POSIXct` 和 `Date` 中的零点指什么时候？

- 问题 11-3
  你会用什么格式化字符串来显示以下日期：完整的月份名，后面跟着四位数的年份？

- 问题 11-4
  如何把一个 `POSIXct` 日期往将来移动一小时？

- 问题 11-5
  使用 `lubridate` 包，考虑以下两个开始于 2016 年 1 月 1 日的时间间隔哪一个后结束，是一年的持续时间，还是一年的周期？

11.9 知识测试：练习

- 练习 11-1
  解析披头士的出生日期，并使用"AbbreviatedWeekday DAYOFMONTH Abbreviated-MonthNameTwoDigitYear"的形式（例如，Wed 09 Oct 40）把它们打印出来。他们的出生日期列于下表。

披头士乐队	成员出生日期
Ringo Starr	1940-07-07
John Lennon	1940-10-09
Paul McCartney	1942-06-18
George Harrison	1943-02-25

[10]

- 练习 11-2

  R 中有一个文件用于查找 Olson 时区的名称，使用编程方式来读取此文件。`?Sys.timezone` 帮助页面中的示例会告诉你方法。查找你所在时区的名称。[10]

- 练习 11-3

  编写一个函数，它以日期为输入参数，并能返回对应于当天的星座。每个星座的日期范围列于下表。[15]

星座	起始日期	结束日期
白羊座	3 月 21 日	4 月 19 日
金牛座	4 月 20 日	5 月 20 日
双子座	5 月 21 日	6 月 20 日
巨蟹座	6 月 21 日	7 月 22 日
狮子座	7 月 23 日	8 月 22 日
处女座	8 月 23 日	9 月 22 日
天秤座	9 月 23 日	10 月 22 日
天蝎座	10 月 23 日	11 月 21 日
射手座	11 月 22 日	12 月 21 日
摩羯座	12 月 22 日	1 月 19 日
水瓶座	1 月 20 日	2 月 18 日
双鱼座	2 月 19 日	3 月 20 日

第二部分

数据分析工作流

第 12 章

获取数据

数据的来源可以有很多。R 内置有许多数据集，而在其他的附件包中能找到更多的数据。R 能从各式各样的来源中读取数据，且支持大量的文件格式。本章将介绍如何从文本文件（包括以逗号或制表符分隔的类似电子表格的格式、XML 和 JSON 文件）、二进制文件（Excel 电子表格和从其他分析软件获取的数据）以及从网站和数据库中导入数据。

12.1 本章目标

阅读本章后，你会了解以下内容：

- 如何访问 R 包中所提供的数据集；
- 如何从文本文件导入数据；
- 如何从二进制文件中导入数据；
- 如何从网站上下载数据；
- 如何从数据库中导入数据。

12.2 内置的数据集

R 的基本分发包中有一个叫 `datasets`，里面全是示例数据集。如果其中有适用于你的研究领域的包，那么你很走运，因为它们非常适合测试代码及探索新的技术。很多其他包也含有数据集。使用 `data` 函数可查看所有你已成功加载了的包的数据集：

```
data()
```

如需更完整的列表，包括已安装的所有包的数据，可使用以下方法：

```
data(package = .packages(TRUE))
```

如果你想访问任意数据集里的数据，只需调用 data 函数，传入数据集的名称及其所在的包名（如果此包已被加载，可省略这个 package 参数）：

```
data("kidney", package = "survival")
```

现在，可以把 kidney 的数据框当成你自己的变量使用：

```
head(kidney)

##   id time status age sex disease frail
## 1  1    8      1  28   1   Other   2.3
## 2  1   16      1  28   1   Other   2.3
## 3  2   23      1  48   2      GN   1.9
## 4  2   13      0  48   2      GN   1.9
## 5  3   22      1  32   1   Other   1.2
## 6  3   28      1  32   1   Other   1.2
```

12.3 读取文本文件

有众多的格式和文本文件标准可用于存储数据。用于存储数据的通用格式为分隔符值（即 CSV 或制表符分隔文件）、可扩展标记语言（XML）、JavaScript 对象表示法（JSON）和 YAML（这是 YAML Ain't Markup Language 的递归表示）。文本数据的其他来源结构会比较松散，例如书本，其中并没有包含任何正式的文本数据（即标准化和可被机器解析的）结构。

将数据存储在文本文件中的主要优点是：它们可被几乎所有的其他数据分析软件或人读取。这使得你的数据能广泛地被重用。

12.3.1 CSV和制表符分隔（Tab-Delimited）文件

矩形（类似电子表格的）数据通常存储在带有分隔符的文件中，特别是逗号分隔值（CSV）和制表符分隔值文件。read.table 函数将读取这些分隔符文件，并将结果存储在一个数据框中。它就是这么简单，只需输入文本文件的路径，即可导入其内容。

RedDeerEndocranialVolume.dlm 是一个以空格符分隔的文件，它包含了一些使用不同技术测量得到的马鹿的颅容积数据。（对于那些对鹿头骨感兴趣的人来说，该方法就是计算机 X 射线断层术，它会用到 Finarelli 方程。它用玻璃珠填充满头骨，然后使用卡钳测量其长度、宽度和高度。在某些情况下，作第二次测量能够知道所采用技术的准确性。我已确认：鹿死很久之后才会将它们的头骨填满珠子！）数据文件可以在 learningr 包的 extdata 文件夹中找到。前几行数据请参见表 12-1 。

表12-1： RedDeerEndocranialVolume.dlm样本中的数据

SkullID	VolCT	VolBead	VolLWH	VolFinarelli	VolCT2	VolBead2	VolLWH2
DIC44	389	375	1484	337			
B11	389	370	1722	377			
DIC90	352	345	1495	328			
DIC83	388	370	1683	377			
DIC787	375	355	1458	328			
DIC1573	325	320	1363	291			
C120	346	335	1250	289	346	330	1264
C25	302	295	1011	250	303	295	1009
F7	379	360	1621	347	375	365	1647

该数据有标题行，所以我们需要给 `read.table` 传递参数 `header = TRUE`。因为并不是每次都会进行第二次测量，所以不是所有行都是完整的。给 `read.table` 传递参数 `fill = TRUE` 会使用 NA 值来代替那些缺失的域。下例中的 `system.file` 函数用于定位包中的文件（在下例中，RedDeerEndocranialVolume.dlm 文件在 `learningr` 包中的 `extdata` 文件夹中）。

```
library(learningr)
deer_file <- system.file(
  "extdata",
  "RedDeerEndocranialVolume.dlm",
  package = "learningr"
)
deer_data <- read.table(deer_file, header = TRUE, fill = TRUE)
str(deer_data, vec.len = 1)      #vec.len 改变了输出的数量

## 'data.frame':        33 obs. of  8 variables:
##  $ SkullID     : Factor w/ 33 levels "A4","B11","B12",..: 14 ...
##  $ VolCT       : int  389 389 ...
##  $ VolBead     : int  375 370 ...
##  $ VolLWH      : int  1484 1722 ...
##  $ VolFinarelli: int  337 377 ...
##  $ VolCT2      : int  NA NA ...
##  $ VolBead2    : int  NA NA ...
##  $ VolLWH2     : int  NA NA ...
```

```
head(deer_data)

##   SkullID VolCT VolBead VolLWH VolFinarelli VolCT2 VolBead2 VolLWH2
## 1   DIC44   389     375   1484          337     NA       NA      NA
## 2     B11   389     370   1722          377     NA       NA      NA
## 3   DIC90   352     345   1495          328     NA       NA      NA
## 4   DIC83   388     370   1683          377     NA       NA      NA
## 5  DIC787   375     355   1458          328     NA       NA      NA
## 6 DIC1573   325     320   1363          291     NA       NA      NA
```

注意，每个列的类已自动确定，行和列的名字也已自动分配。列名（默认情况下）必须

是有效的变量名（通过使用 `make.names`），如果不提供行名那么行将就会按 1、2、3 编号，以此类推。

有很多参数可以用来指定如何读取该文件，其中最重要的可能是 `sep`，它决定了使用哪个字符作为字段之间的分隔符。`nrow` 可以指定读取数据的行数，而 `skip` 决定跳过文件开始的多少行。更多高级选项包括：覆盖默认的行名、列名和类，指定输入文件的字符编码，以及输入的字符串格式的列如何声明。

有几个 `read.table` 的包装函数使用起来比较方便。`read.csv` 分隔符默认设置为逗号，并假设数据有标题行。`read.csv2` 是它的欧洲表亲，它使用逗号作为小数位，并用分号作为分隔符。同样地，`read.delim` 和 `read.delim2` 将分别使用句号或逗号作为小数位来导入制表符分隔的文件。

早在 2008 年 8 月，在英国的洛斯托夫特，环境、渔业和水产养殖科学中心（CEFAS）的科学家把贴上标签的压力和温度传感器放在一种棕色螃蟹身上，并把它们投入北海。螃蟹自在地游荡了一年之后[1]，渔民抓住了它，并把标签送回了 CEFAS。

从该标签中获取的数据以及一些元数据都存储在一个 CSV 文件中。文件的前几行是这样的：

Comment :- clock reset to download data	
The following data are the ID block contents	
Firmware Version No	2
Firmware Build Level	70
The following data are the Tag notebook contents	
Mission Day	405
Last Deployment Date	08/08/2008 09:55:00
Deployed by Host Version	5.2.0
Downloaded by Host Version	6.0.0
Last Clock Set Date	05/01/2010 10:34:00
The following data are the Lifetime notebook contents	
Tag ID	A03401
Pressure Range	10
No of sensors	2

在这种情况下，我们不能仅调用 `read.csv` 就把所有东西都读取出来，因为不同的数据块中所含的字段数量不同，而且每个字段也确实不同。我们需要使用 `read.csv` 中的 `skip` 和 `nrow` 参数指定要读取文件中哪些位置：

```
crab_file <- system.file(
  "extdata",
  "crabtag.csv",
  package = "learningr"
```

注 1：具体就是从北海东部德国附近迁移到北海西部英国附近。

```
)
(crab_id_block <- read.csv(
  crab_file,
  header = FALSE,
  skip = 3,
  nrow = 2
))

##                     V1 V2
## 1 Firmware Version No 2
## 2 Firmware Build Level 70

(crab_tag_notebook <- read.csv(
crab_file,
header = FALSE,
skip = 8,
nrow = 5
))


##                             V1                  V2
## 1                  Mission Day                 405
## 2       Last Deployment Date 08/08/2008 09:55:00
## 3   Deployed by Host Version               5.2.0
## 4 Downloaded by Host Version               6.0.0
## 5        Last Clock Set Date 05/01/2010 10:34:00

(crab_lifetime_notebook <- read.csv(
  crab_file,
  header = FALSE,
  skip = 15,
  nrow = 3
))

##                 V1     V2
## 1         Tag ID A03401
## 2 Pressure Range     10
## 3  No of sensors      2
```

colbycol 和 sqldf 包中的函数可用于把 CSV 文件的部分数据读取到 R 中。如果你并不需要所有列或所有行，这能加快速度。

在导入此类文件时，你可以使用 scan 函数来真正地从底层进行控制，read.table 也是基于此函数实现的。通常情况下，scan 应该被避免使用，但它对于处理格式奇怪的和非标准的文件非常有用。

如果你的数据是从另一种语言中导入的，那么可能需要把 na.strings 参数传递给 read.table。对于从 SQL 导出的数据，则使用 na.strings = "NULL"。对于从 SAS 或 Stata 导出的数据，使用 na.strings = "."。从 Excel 中导出的数据，使用 na.strings = c("", "#N/A", "#DIV/0!", "#NUM!")。

与此相反的任务是写入文件，这通常比读取文件要更简单，因为你无需担心读取文件时出现的各种怪事——通常人们想创造一些标准的东西。很明显，`write.table` 和 `write.csv` 分别对应着 `read.table` 和 `read.csv` 的读操作。

这两个函数都需要一个数据框和写入文件的路径作为参数。它们还提供了几个选项以自定义输出（例如：是否包含列名，应使用什么样的输出文件字符编码）：

```
write.csv(
  crab_lifetime_notebook,
  "Data/Cleaned/crab lifetime data.csv",
  row.names = FALSE,
  fileEncoding = "utf8"
)
```

12.3.2 非结构化文本文件

不是所有的文本文件都像定界符文件那样有一个定义良好的结构。如果文件的结构松散，更简单的做法是：先读入文件中的所有文本行，再对其内容进行分析或操作。`readLines`（注意大写字母 L）就提供了这种方法。它接受一个文件路径（或文件连接）和一个可选的最大行数作为参数来读取文件。这里将导入莎士比亚的《暴风雨》的古登堡计划版本：

```
text_file <- system.file(
  "extdata",
  "Shakespeare's The Tempest, from Project Gutenberg pg2235.txt",
  package = "learningr"
)
the_tempest <- readLines(text_file)
the_tempest[1926:1927]

## [1] " Ste. Foure legges and two voyces; a most delicate"
## [2] "Monster: his forward voyce now is to speake well of"
```

`writeLines` 用于执行与 `readLines` 相反的操作。它写入文件时需要一个字符向量和文件作为输入参数：

```
writeLines(
  rev(text_file), #rev 执行向量的相反操作
  "Shakespeare's The Tempest, backwards.txt"
)
```

12.3.3 XML和HTML文件

XML 文件被广泛地用于存储嵌套数据。许多标准的文件类型和协议都基于它，例如：用于提供新闻资料的 RSS（Really Simple Syndication），用于跨计算机网络传递结构化数据的 SOAP（Simple Object Access Protocol），以及网页中常用的 XHTML。

R 的基本包没有读取 XML 文件的能力，不过 `XML` 包已被某 R 核心成员开发出来了。现在

就安装它吧！

```
install.packages("XML")
```

当你导入一个 XML 文件时，该 XML 包将提供两种选择以存储结果：一是利用内部节点（即使用 C 代码来存储对象，这是默认值），或使用 R 节点。通常，你应使用内部节点来存储，因为这样就能用 XPath（马上就会谈到它）来查询节点树。

有几个函数可用于导入 XML 数据，如 xmlParse 和其他一些使用稍不同的默认值的包装函数：

```
library(XML)
xml_file <- system.file("extdata", "options.xml", package = "learningr")
r_options <- xmlParse(xml_file)
```

使用内部节点的一个问题是，str 和 head 等汇总函数不能和它们一起使用。要使用 R- 级的节点，请设置 useInternalNodes = FALSE（或使用 xmlTreeParse，它会默认设置此项属性）：

```
xmlParse(xml_file, useInternalNodes = FALSE)
xmlTreeParse(xml_file)      # 作用相同
```

XPath 是一种用于查询 XML 文档的语言，它能基于某些过滤规则寻找到相应的节点。在下例中，我们将在文档 // 中到处寻找命名为 variable 的结点，此结点 [] 的 name 属性 @ 包含 contains 了 warn 字符串。

```
xpathSApply(r_options, "//variable[contains(@name, 'warn')]")

## [[1]]
## <variable name="nwarnings" type="numeric">
## <value>50</value>
## </variable>
##
## [[2]]
## <variable name="warn" type="numeric">
## <value>0</value>
## </variable>
##
## [[3]]
## <variable name="warning_length" type="numeric">
## <value>1000</value>
## </variable>
```

这种查询在提取网页数据中非常有用。正如你所料，htmlParse 和 htmlTreeParse 是用于 HTML 页面导入的函数，它们的行为方式基本一样。 XML 格式在序列化（也即存储）对象时非常有用，这种格式可被大多数其他软件读取。XML 包没有提供序列化的功能，但你可以使用 Runiversal 包中的 makexml 函数来完成它。Options.xml 文件由以下代码创建：

```
library(Runiversal)
ops <- as.list(options())
cat(makexml(ops), file = "options.xml")
```

12.3.4 JSON和YAML文件

XML 的主要问题是它太冗长了，且你需要显式地指定数据的类型（它在默认情况下不能区分字符串和数字），这就使得它更冗长了。如果文件大小很重要（例如，当你要在网络上传输大量数据集时），信息过于冗余就成了问题。

于是，有人发明了 YAML 和它的子集 JSON 来解决这些问题。它们特别适合于通过网络传输大量数据集，尤其是数字数据和数组。JSON 是 Web 应用程序彼此之间传递数据的事实标准。

有两个包可用于处理 JSON 数据：`RJSONIO` 和 `rjson`。在很长一段时间内，rjson 都存在性能问题，因此唯一值得推荐的包是 `RJSONIO`。不过，性能的问题现在已经修复，所以它也算是一个备选包。在大多数情况下，你使用哪个包都可以。只有当你遇到格式不正确或非标准的 JSON 时会看出差别。

在读入不正确的 JSON 时，`RJSONIO` 一般比 `rjson` 更宽容。这是否是好事取决于你所使用的场景。如果你想简单地导入 JSON 数据，`RJSONIO` 就是最好的选择。如果你想对有问题的 JSON 数据的保持警觉（或许是你的同事生成的——我肯定，你绝不会生成有问题的 JSON），那么 `rjson` 就是最好的。

幸好，在这两个包中读取和写入 JSON 数据的函数名基本相同，所以很容易在它们之间切换。在下例中，双冒号 `::` 用于把相同名字的函数从不同的包中分别出来（如果只加载两个包中的一个，就不需要双冒号）：

```
library(RJSONIO)
library(rjson)
jamaican_city_file <- system.file(
  "extdata",
  "Jamaican Cities.json",
  package = "learningr"
)
(jamaican_cities_RJSONIO <- RJSONIO::fromJSON(jamaican_city_file))

## $Kingston
## $Kingston$population
## [1] 587798
##
## $Kingston$coordinates
## longitude  latitude
##     17.98     76.80
##
##
## $`Montego Bay`
## $`Montego Bay`$population
## [1] 96488
##
## $`Montego Bay`$coordinates
```

```
## longitude  latitude
##     18.47     77.92
```

```
(jamaican_cities_rjson <- rjson::fromJSON(file = jamaican_city_file))
```

```
## $Kingston
## $Kingston$population
## [1] 587798
##
## $Kingston$coordinates
## $Kingston$coordinates$longitude
## [1] 17.98
##
## $Kingston$coordinates$latitude
## [1] 76.8
##
##
##
## $`Montego Bay`
## $`Montego Bay`$population
## [1] 96488
##
## $`Montego Bay`$coordinates
## $`Montego Bay`$coordinates$longitude
## [1] 18.47
##
## $`Montego Bay`$coordinates$latitude
## [1] 77.92
```

请注意，RJSONIO 把每个城市的坐标简化为一个向量。这种功能可通过 simplify = FALSE 来关闭，这样产生的对象就和 rjson 产生的完全一样。

有点讨厌的是，JSON 的规范不允许无穷值或 NaN 值，而且它对缺失数的定义比较模糊。这两个包处理这些值的方式有所不同：RJSONIO 把 NaN 和 NA 映射为 JSON 的 null，但保留正负无穷；而 rjson 会把所有这些值都转换为字符串：

```
special_numbers <- c(NaN, NA, Inf, -Inf)
RJSONIO::toJSON(special_numbers)
```

```
## [1] "[ null, null,    Inf,   -Inf ]"
```

```
rjson::toJSON(special_numbers)
```

```
## [1] "[\"NaN\",\"NA\",\"Inf\",\"-Inf\"]"
```

因为这两种方法都用于处理备受限制的 JSON 规范，所以如果你发现需要大量地处理这些特殊数字类型（或想在你的数据对象中加些评论），那么最好还是使用 YAML。在 yaml 包中有两个函数能导入 YAML 数据：yaml.load 接受一个 YAML 的字符串，并将其转换为一个 R 对象；yaml.load_file 也一样，不过它把输入的字符串作为包含 YAML 文件的路径处理：

```
library(yaml)
yaml.load_file(jamaican_city_file)

## $Kingston
## $Kingston$population
## [1] 587798
##
## $Kingston$coordinates
## $Kingston$coordinates$longitude
## [1] 17.98
##
## $Kingston$coordinates$latitude
## [1] 76.8
##
##
##
## $`Montego Bay`
## $`Montego Bay`$population
## [1] 96488
##
## $`Montego Bay`$coordinates
## $`Montego Bay`$coordinates$longitude
## [1] 18.47
##
## $`Montego Bay`$coordinates$latitude
## [1] 77.92
```

`as.yaml` 则执行相反的任务，它会把 R 对象转换为 YAML 字符串。

12.4 读取二进制文件

很多软件都把它们的数据存储为二进制格式（有一些是专有的，有一些则符合公开定义的标准）。二进制格式通常要比它们的文本格式要小，因此使用二进制格式的性能可能会更好，不过这是以可读性为代价的。

许多二进制文件的格式都是专有的，这违背了自由软件的原则。如果你可以选择，最好不要使用这种格式，以免数据被锁定在一个你缺乏控制权的平台上。

12.4.1 读取Excel文件

Microsoft Excel 是目前世界上最流行的电子表格程序，也很可能是世界上最流行的数据分析工具。可惜，它的文档格式 XLS 和 XLSX 与其他软件的兼容性不好，尤其是对于那些非 Windows 平台的软件。这意味着，你需要做一些试验才能找到合适的配置，使你所选择的操作系统能与 Excel 文件兼容。

`xlsx` 是基于 Java 的跨平台包，这就是说，至少在理论上，它可以在任何系统上读取任何 Excel 文件。它提供了读取 Excel 文件的函数：使用 `read.xlsx` 和 `read.xlsx2` 导入电子表

格，它们分别在 R 和 Java 中做了更多的处理工作。有两种选择其实很多余，你当然会选择 `read.xlsx2`，因为它的速度更快且底层的 Java 库也比 R 程序更加成熟。

下例显示了环法自行车赛拉普德兹山地段的最佳成绩，以及每个赛车手是否因服用禁药而被判有罪的记录。`colClasses` 参数将决定数据框中每列的类型。这并非是强制的，但这样做可以免去日后再处理数据的麻烦：

```
library(xlsx)
bike_file <- system.file(
  "extdata",
  "Alpe d'Huez.xls",
  package = "learningr"
)
bike_data <- read.xlsx2(
  bike_file,
  sheetIndex = 1,
  startRow = 2,
  endRow = 38,
  colIndex = 2:8,
  colClasses = c(
    "character", "numeric", "character", "integer",
    "character", "character", "character"
  )
)
head(bike_data)
```

```
##      Time NumericTime            Name Year   Nationality DrugUse
## 1 37' 35"       37.58   Marco Pantani 1997         Italy       Y
## 2 37' 36"       37.60 Lance Armstrong 2004 United States       Y
## 3 38' 00"       38.00   Marco Pantani 1994         Italy       Y
## 4 38' 01"       38.02 Lance Armstrong 2001 United States       Y
## 5 38' 04"       38.07   Marco Pantani 1995         Italy       Y
## 6 38' 23"       38.38     Jan Ullrich 1997       Germany       Y
##                                                 Allegations
## 1 Alleged drug use during 1997 due to high haematocrit levels.
## 2         2004 Tour de France title stripped by USADA in 2012.
## 3 Alleged drug use during 1994 due to high haematocrit levels.
## 4         2001 Tour de France title stripped by USADA in 2012.
## 5 Alleged drug use during 1995 due to high haematocrit levels.
## 6                 Found guilty of a doping offense by the CAS.
```

`xlsReadWrite` 是 `xlsx` 包的另一个选择，但它现在只适用于 Windows 下的 32 位 R 系统。还有一些其他的包能与 `Excel` 一起工作。例如，`RExcel` 和 `excel.link` 使用 COM 连接从 R 中控制 Excel，`WriteXLS` 使用 Perl 写入 Excel 文件。`gnumeric` 包提供了读取 Gnumeric 电子表格的函数。

与 `read.xlsx2` 相对应的函数是（你猜对了）`write.xlsx2`。它的工作方式与 `write.csv` 相同，它的参数为一个数据框和文件名。除非你真的需要使用 Excel 电子表格，否则为了方便，你最好还是把数据保存为文本格式。所以请小心使用。

12.4.2 读取SAS、Stata、SPSS和MATLAB文件

如果你正在与其他组织的统计人员合作，他们可能会向你发送一些从其他统计包中生成的文件。`foreign` 包中包含了使用 `read.ssd` 读取 SAS 永久数据集[2]（SAS7BDAT 文件）、使用 `read.dta` 读取 Stata 的 DTA 文件、使用 `read.spss` 读取 SPSS 数据文件的方法等。这些文件都可使用 `write.foreign` 写入文件。

MATLAB 的二进制数据文件（4 级及 5 级）可以使用 `R.matlab` 包中的 `readMat` 和 `writeMat` 分别读写。

12.4.3 读取其他文件类型

R 可读取多种其他类型的文件数据。

它可以使用 `h5r` 包（以及在 Bioconductor 中的 `rdhf5` 包）读取分层数据格式 V5 [HDF5] 文件，亦可使用 `ncdf` 包读取网络通用数据格式 [NetCDF]。

它可以使用 `maptools` 和 `shapefiles` 包读取 ESRI 公司的 ArcGIS 空间数据文件（以及使用 `RArcInfo` 包读取老版本的 ArcInfo 文件）。

它可以通过 `jpeg`、`png`、`tiff`、`rtiff` 和 `readbitmap` 包读取光栅图像格式。

它可以使用 Bioconductor 中的包来读取各种基因组数据格式。其中最值得注意的是，它可以通过 `RPPanalyzer` 包读取 GenePix GPR 文件（也称为轴突文本文件）；通过 `vcf2geno` 包读取基因序列变异的 VCF（Variant Call Format）；通过 `rbamtools`（它提供了一个 SAMtools 的接口）读取二进制序列比对数据；通过 `lxb` 包读取 Luminex bead array assay 文件。

最后，还有很多各式各样的格式散布于其他包中。不完全名单包括：用于 MRI 图像的 `4dfp` 和 `tractor.base` 包，用于 WaveMetrics Igor 二进制格式文件的 `IgorR` 包，用于 GENEActiv 观看加速度计数据显示的 `GENEAread` 包，用于 INRO 软件 EMME v2 数据库文件的 `emme2` 包，用于 SEER 癌症数据集（http://seer.cancer.gov/seerstat）的 `SEER2R` 包，用于 Google Protocol Buffer（http://code.google.com/p/protobuf）的 `rprotobuf` 包，用于 Bruker flex 格式（http://strimmerlab.org/software/maldiquant/）中质谱数据的 `readBrukerFlexData` 包，用于在 Models-3 文件中社区多尺度空气质量模式的 M3 包，以及用于世界生育率调查的 `Read.isi` 包。

虽然以上很多的包对于大多数人来说完全没用，但 R 在广泛的领域中有众多的专门应用——这一点是相当震撼的。

注 2：目前还不能把老的 SAS SD2 文件导入到 R。处理这种格式的最简方法就是用免费的 SAS 通用浏览器（http://bit.ly/ldqSmrB）打开它，并重新保存为 CSV。

12.5 Web数据

互联网上包含了大量数据，但如果你想手动访问网站下载那里的数据文件，然后再把它从硬盘中读取到 R 中，这一系列动作需要大量的手动操作（且也不易扩展）。

幸好，R 有很多种方式可以从网络源导入数据并以编程的方式检索数据，这就使数据处理的工作事半功倍。

12.5.1 拥有API的网站

有一些包可通过网站的 API 接口直接下载数据到 R 系统。例如，世界银行已在其网站上公开了它的世界发展指标数据，`WDI` 包可以让你轻松地在 R 中将这些数据导入。要运行以下例子，你需要先安装 `WDI` 包：

```
install.packages("WDI")

library(WDI)

# 列出所有可用的数据集
wdi_datasets <- WDIsearch()
head(wdi_datasets)

##      indicator
## [1,] "BG.GSR.NFSV.GD.ZS"
## [2,] "BM.KLT.DINV.GD.ZS"
## [3,] "BN.CAB.XOKA.GD.ZS"
## [4,] "BN.CUR.GDPM.ZS"
## [5,] "BN.GSR.FCTY.CD.ZS"
## [6,] "BN.KLT.DINV.CD.ZS"
##      name
## [1,] "Trade in services (% of GDP)"
## [2,] "Foreign direct investment, net outflows (% of GDP)"
## [3,] "Current account balance (% of GDP)"
## [4,] "Current account balance excluding net official capital grants (% of GDP)"
## [5,] "Net income (% of GDP)"
## [6,] "Foreign direct investment (% of GDP)"

# 取其中一个
wdi_trade_in_services <- WDI(
   indicator = "BG.GSR.NFSV.GD.ZS"
)
str(wdi_trade_in_services)

## 'data.frame':  984 obs. of  4 variables:
## $ iso2c             : chr  "1A" "1A" "1A" ...
## $ country           : chr  "Arab World" "Arab World" "Arab World" ...
## $ BG.GSR.NFSV.GD.ZS : num  17.5 NA NA NA ...
## $ year              : num  2005 2004 2003 2002 ...
```

`SmarterPoland` 包提供了类似的功能，它封装了波兰政府的数据。使用 `quantmod` 包可以访

问股票行情（默认是雅虎的数据，也可以选择其他几个数据源）：

```
library(quantmod)
# 如果你正在使用 0.5.0 之前的版本，那么请设置以下选项，
# 或者把参数 auto.assign = FALSE 传给 getSymobols。
options(getSymbols.auto.assign = FALSE)
microsoft <- getSymbols("MSFT")

head(microsoft)

##            MSFT.Open MSFT.High MSFT.Low MSFT.Close MSFT.Volume
## 2007-01-03     29.91     30.25    29.40      29.86    76935100
## 2007-01-04     29.70     29.97    29.44      29.81    45774500
## 2007-01-05     29.63     29.75    29.45      29.64    44607200
## 2007-01-08     29.65     30.10    29.53      29.93    50220200
## 2007-01-09     30.00     30.18    29.73      29.96    44636600
## 2007-01-10     29.80     29.89    29.43      29.66    55017400
##            MSFT.Adjusted
## 2007-01-03         25.83
## 2007-01-04         25.79
## 2007-01-05         25.64
## 2007-01-08         25.89
## 2007-01-09         25.92
## 2007-01-10         25.66
```

`TwitterR` 包能够访问 Twitter 账号及其微博信息。只需要事先配置好（由于 Twitter 的 API 需要你创建一个应用程序，并使用 OAuth 进行注册。请阅读此包的安装说明），此包就能非常容易地导入 Twitter 的数据并作进一步的网络分析，或假装工作而实际上只是在刷微博。

12.5.2　抓取网页

R 有内置的 web 服务器，所以某些需要读取数据的函数默认是带有网络访问功能的。`read.table`（及其衍生函数，例如 `read.csv`）接受一个 URL 作为参数（而不是一个本地文件），它会在导入数据之前将副本下载到一个临时文件中。例如，经济研究员 Justin Rao 的网站有从 2002 年到 2008 年间的 NBA 工资数据：

```
salary_url <- "http://www.justinmrao.com/salary_data.csv"
salary_data <- read.csv(salary_url)
str(salary_data)
```

由于通过互联网访问一个大文件时速度可能会很慢，如果经常使用该文件，更好的策略是用 `download.file` 对要下载的文件创建一个本地副本，然后再导入：

```
salary_url <- "http://www.justinmrao.com/salary_data.csv"
local_copy <- "my local copy.csv"
download.file(salary_url, local_copy)
salary_data <- read.csv(local_copy)
```

其他更高级的网页访问方法可以在 `RCurl` 包中找到，它能访问 `libcurl` 网络客户端接口程序库。如果你的数据包含在 HTML 或 XML 页面中，而不是刚好被放在网络上的标准数据格式（如 CSV）时，这一点尤其有用。

下例将从美国海军天文台时间服务部的网站上检索几个时区中的当前日期和时间。`getURL` 函数将访问页面并将其内容以字符串的形式返回：

```
library(RCurl)
time_url <- "http://tycho.usno.navy.mil/cgi-bin/timer.pl"
time_page <- getURL(time_url)
cat(time_page)

## <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final"//EN>
## <html>
## <body>
## <TITLE>What time is it?</TITLE>
## <H2> US Naval Observatory Master Clock Time</H2> <H3><PRE>
## <BR>Jul. 17, 20:43:37 UTC            Universal Time
## <BR>Jul. 17, 04:43:37 PM EDT Eastern Time
## <BR>Jul. 17, 03:43:37 PM CDT Central Time
## <BR>Jul. 17, 02:43:37 PM MDT Mountain Time
## <BR>Jul. 17, 01:43:37 PM PDT Pacific Time
## <BR>Jul. 17, 12:43:37 PM AKDT        Alaska Time
## <BR>Jul. 17, 10:43:37 AM HAST        Hawaii-Aleutian Time
## </PRE></H3><P><A HREF="http://www.usno.navy.mil"> US Naval Observatory</A>
##
## </body></html>
```

下一步几乎总是使用 `XML` 包中的 `htmlParse`（或相关函数）解析页面。这使你可以提取其有用的节点。在下例中，以 `\n`（换行符）为分隔符将得到每个时间线，使用 `\t`（制表符）分隔则能获得时间 / 时区对：

```
library(XML)
time_doc <- htmlParse(time_page)
pre <- xpathSApply(time_doc, "//pre")[[1]]
values <- strsplit(xmlValue(pre), "\n")[[1]][-1]
strsplit(values, "\t+")

## [[1]]
## [1] "Jul. 17, 20:43:37 UTC" "Universal Time"
##
## [[2]]
## [1] "Jul. 17, 04:43:37 PM EDT" "Eastern Time"
##
## [[3]]
## [1] "Jul. 17, 03:43:37 PM CDT" "Central Time"
##
## [[4]]
## [1] "Jul. 17, 02:43:37 PM MDT" "Mountain Time"
##
## [[5]]
```

```
## [1] "Jul. 17, 01:43:37 PM PDT" "Pacific Time"
##
## [[6]]
## [1] "Jul. 17, 12:43:37 PM AKDT" "Alaska Time"
##
## [[7]]
## [1] "Jul. 17, 10:43:37 AM HAST" "Hawaii-Aleutian Time"
```

httr 包基于 RCurl，它能提供了更好的语法，使我们的工作更加方便。在 httr 中，相当于 RCurl 包中 getURL 的函数是 GET，content 函数能在检索页面内容的同时解析它。在下例中，我们将传入 useInternalNodes = TRUE 来模仿 htmlParse 的行为，并重复上例中的行为：

```
library(httr)
time_page <- GET(time_url)
time_doc <- content(page, useInternalNodes = TRUE)
```

12.6 访问数据库

如果数据需要被多人访问，那最好将它们存储在一个关系数据库中。有许多数据库管理系统（DBMS）用于关系数据库管理，R 也可与所有常见的数据库相连接。DBI 包为访问 DBMS 提供了统一的语法——目前的 SQLite、MySQL/MariaDB、PostgreSQL 和 Oracle 都能支持，它还提供了一个封装了 JDBC（Java Database Connectivity）API 的函数。（连接到 SQL Server 须使用不同的系统，我们将在下面看到。）

要连接到 SQLite 数据库，首先你须安装并加载 DBI 包及后端的 RSQLite 包：

```
library(DBI)
library(RSQLite)
```

然后，定义数据库驱动程序的类型是“SQLite”，并通过命名该文件来设置到数据库的连接：

```
driver <- dbDriver("SQLite")
db_file <- system.file(
  "extdata",
  "crabtag.sqlite",
  package = "learningr"
)
conn <- dbConnect(driver, db_file)
```

对于 MySQL 数据库来说，则需加载 RMySQL 包，并设置驱动器类型为“MySQL”：

```
driver <- dbDriver("MySQL")
db_file <- "path/to/MySQL/database"
conn <- dbConnect(driver, db_file)
```

对于 PostgreSQL、Oracle 和 JDBC 来说，它们分别需要 PostgreSQL、ROracle 和 RJDBC 包，它们的数据库名也是其驱动程序的名字，与 SQLite 和 MySQL 一样。

要从数据库中检索数据，你得写一些 SQL 查询语句，并使用 dbGetQuery 把它发送到数据库中。在下例中，SELECT * FROM IdBlock 的意思是：从 IdBlock 表中取得所有列的数据[3]：

```
query <- "SELECT * FROM IdBlock"
(id_block <- dbGetQuery(conn, query))

##   Tag ID Firmware Version No Firmware Build Level
## 1 A03401                   2                   70
```

接着，在完成数据库操作之后，断开和卸载驱动程序以完成清理工作：

```
dbDisconnect(conn)

## [1] TRUE

dbUnloadDriver(driver)

## [1] TRUE
```

很容易一不小心就忘记关闭连接，特别是当你的连接发生错误时。要避免这种情况，一种方法是把你的代码封装到一个函数中，并在 on.exit 中确保清理代码始终会被执行。on.exit 在其父函数退出时会运行 R 的代码，不管它是否正确地完成或发生了错误。我们可以使用更安全的代码重写之前的范例，如下：

```
query_crab_tag_db <- function(query)
{
  driver <- dbDriver("SQLite")
  db_file <- system.file(
    "extdata",
    "crabtag.sqlite",
    package = "learningr"
  )
  conn <- dbConnect(driver, db_file)
  on.exit(
    {
      # 此代码在函数之后运行，
      # 即使抛出一个错误
      dbDisconnect(conn)
      dbUnloadDriver(driver)
    }
  )
  dbGetQuery(conn, query)
}
```

向函数传入一些 SQL 代码来查询螃蟹标签的数据库：

```
query_crab_tag_db("SELECT * FROM IdBlock")

##   Tag ID Firmware Version No Firmware Build Level
## 1 A03401                   2                   70
```

注 3：任何称职的数据库管理员都会告诉你，SELECT* 这种写法完全是偷懒，在你把代码随意地运行在服务器上之前，应该非常明确地指出你需要哪些列。非常感谢。

在此例中，DBI 包提供了一个很实用的函数，使我们不用自己写 SQL 代码。dbReadTable 做你所期望的事：当忘记你想要的表的名称时，它将从已连接的数据库中读取表（使用 dbListTables (conn)）：

```
dbReadTable(conn, "idblock")

##   Tag ID Firmware Version No Firmware Build Level
## 1 A03401                   2                   70

## [1] TRUE

## [1] TRUE
```

如果你的数据库没有出现在以上列出的类型之中，可以使用替代的 RODBC 包，它使用 ODBC 数据库连接——当连接到 SQL Server 或 Access 数据库时，这非常有用。与 DBI 相比，这些函数有着不同的名称，但原理非常相似。设置电脑上的 ODBC 数据源（通过 Windows 下的控制面板，或在“开始”菜单中搜索“ODBC”），R 就可连接上。图 12-1 显示了 Windows 7 上的 OBDC 注册向导。

然后，调用 odbcConnect 连接，使用 sqlquery 运行查询，以及 odbcClose 在最后进行清理：

```
library(RODBC)
conn <- odbcConnect("my data source name")
id_block <- sqlQuery(conn, "SELECT * FROM IdBlock")
odbcClose(conn)
```

从 R 中访问 NoSQL 数据库（即 Not only SQL 的简称。它是轻量级的数据存储仓库，比起传统的 SQL 关系数据库更易扩展）的方法不太成熟。可以通过 RMongo 或 rmongodb 包访问 MongoDB，使用 RCassandra 包访问 Cassandra 数据库，以及 R4CouchDB 包访问 CouchDB[还没出现在 CRAN 上，但可以在 GitHub 下载] 。

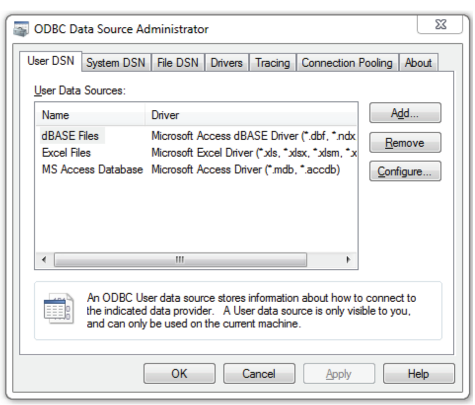


图 12-1：使用 ODBC 数据源管理器注册 ODBC 数据源

12.7 小结

- R 所提供的数据集或包可以从 `data` 函数中取得。
- 可以从很多外部数据源中把数据导入 R。
- `read.table` 及其变种函数能读取矩形数据。
- `readLines` 能读取那些非标准格式的文本文件。
- `XML` 包能读取 HTML 和 XML 数据。
- `RJSONIO`、`rjson` 和 `yaml` 包能读取 JSON/YAML 文件。
- 有很多的包可用于读取 Excel 文件，例如 `xlsx`。
- `foreign` 能从统计软件中读取数据。
- 有大量的包可以用来操作数据库，例如 `DBI` 和 `RODBC`。

12.8 知识测试：问题

- 问题 12-1
  如何在你的机器上找到所有内置于 R 的数据集和包？

- 问题 12-2
  `read.csv` 和 `read.csv2` 函数之间的区别是什么？

- 问题 12-3
  你将如何把数据从 Excel 电子表格中导入到 R 中？

- 问题 12-4
  你将如何导入一个从互联网上找到的 CSV 文件数据？

- 问题 12-5
  `DBI` 包为几个数据库管理系统提供了一个一致的接口。请问哪些系统能被支持？

12.9 知识测试：练习

- 练习 12-1
  在 `learning` 包的 extdata 文件夹中，有一个名为 hafu.csv 的文件，其中有关于混血漫画中的人物数据。请将数据导入到一个数据框中。[5]

- 练习 12-2
  也是在 `learning` 包的 extdata 文件夹中，有一个名为多重耐药淋病的 Excel 文件 infection.xls。从第一个（也是唯一的）表中将数据导入到数据框中。提示：如果你先在电子表格程序中查看该文件会更容易些。`LibreOffice` 包是免费，能轻而易举地完成这个任务。[10]

- 练习 12-3
  从本章中所描述的螃蟹标签 SQLite 数据库中，把 `Daylog` 表的内容导入到一个数据框中。[10]

第 13 章

数据清理和转换

无论给你什么格式的数据，它似乎总和你所要的不是同一回事；而且，无论是谁给你的，它几乎总是脏数据。清理和转换数据可能不是数据分析中有趣的部分，但你却不得不为此花费大量的时间。幸好，R 有多种可供选择的工具来帮你完成这些任务。

13.1　本章目标

阅读本章后，你会了解以下内容：

- 如何对字符串进行操作和清理类别变量；
- 如何抽取数据框的子集以及转换数据框；
- 如何把数据框的形状由宽变长，然后再改回去；
- 理解分类和排序。

13.2　清理字符串

早在第 7 章，我们了解到一些简单的字符串操作任务，如使用 `paste` 把字符串合并在一起，以及使用 `substring` 提取部分字符串等。

一个非常普遍的问题是：逻辑值何时会被编码成 R 不理解的值。在 `alpe_d_huez` 循环数据集中，`DrugUse` 列（指出每个车手是否曾被指控服用违禁药）中数值被编码为“Y”和“N”，而不是 `TRUE` 或 `FALSE`。对于这种简单的匹配关系，我们可以直接使用正确的逻辑值替换掉每个字符串：

```
yn_to_logical <- function(x)
{
  y <- rep.int(NA, length(x))
  y[x == "Y"] <- TRUE
  y[x == "N"] <- FALSE
  y
}
```

默认把值设为 NA 可以让我们处理那些不能匹配“Y”或“N”的字符串。我们可以显式地调用函数：

```
alpe_d_huez$DrugUse <- yn_to_logical(alpe_d_huez$DrugUse)
```

这种直接将一个字符串替换为另一个的方法，对于那些需要替换许多字符串的来说扩展性不是很好。如果你有一万个可能的输入，那么使用一个函数来逐个替换会很难避免书写出错，这种代码也很难维护。

幸运的是，有其他更巧妙的方法可以相对容易地检测、提取和替换掉与模式相匹配的部分字符串。R 有一系列（大致上）基于 Unix grep 工具的内置函数能处理这些任务。它们接受一个要操作的字符串以及要匹配的正则表达式。正如在第 1 章所说的，正则表达式是一种模式，它能非常灵活地描述字符串的内容。在匹配诸如电话号码或电子邮件地址这种复杂的字符串数据类型时，它们非常有用[1]。

grep、grepl 和 regexpr 函数都能找到与模式相匹配的字符串，sub 和 gsub 函数能替换匹配的字符串。在经典的 R 风格中，这些函数都是无比准确和非常强大的，但由于历史的原因，它的命名、参数的顺序和返回值都比较奇怪。幸好，就像 plyr 为 apply 函数，lubridate 为日期－时间函数提供了一致的封装一样，stringr 包对字符串操作函数也提供了一致的封装。不同之处在于，偶尔需要使用基本的 apply 函数或日期－时间函数时，stringr 已足够先进，你根本无须再使用 grep。因此，浏览一下 ?grep 的帮助页面即可，无需投入太多精力。

下例使用 learning 包中的 english_monarchs 数据集。它包含了从后罗马时代（5 世纪）英格兰被分裂为七王国直到 13 世纪初英国接管了爱尔兰这段时期的统治者的名字和日期：

```
data(english_monarchs, package = "learningr")
head(english_monarchs)

##       name     house start.of.reign end.of.reign      domain
## 1    Wehha Wuffingas             NA          571 East Anglia
## 2    Wuffa Wuffingas            571          578 East Anglia
## 3   Tytila Wuffingas            578          616 East Anglia
## 4  Rædwald Wuffingas            616          627 East Anglia
## 5 Eorpwald Wuffingas            627          627 East Anglia
```

注 1：参见 assertive 包为此预装的一些正则表达式。

```
## 6 Ricberht Wuffingas           627         630 East Anglia
##   length.of.reign.years reign.was.more.than.30.years
## 1                    NA                           NA
## 2                     7                        FALSE
## 3                    38                         TRUE
## 4                    11                        FALSE
## 5                     0                        FALSE
## 6                     3                        FALSE
```

历史的问题之一是它的数据实在太多了。幸好，古怪或杂乱的数据会将你引向历史中有意思的那部分，即我们可以压缩数据而聚焦到有趣的部分。例如，尽管英格兰地区有七个王国，但它们的边界却极不确定，有时一个王国会征服另一个。我们可以通过在 domain 列中搜索逗号找到这些交叉点。为了检测出相关的模式，我们使用 str_detect 函数。fixed 函数告诉 str_detect：我们正在寻找一个固定字符串（逗号）而非正则表达式。str_detect 将返回一个可用于索引的逻辑向量：

```
library(stringr)
multiple_kingdoms <- str_detect(english_monarchs$domain, fixed(","))
english_monarchs[multiple_kingdoms, c("name", "domain")]
```

```
##                         name                   domain
## 17                      Offa       East Anglia, Mercia
## 18                      Offa East Anglia, Kent, Mercia
## 19         Offa and Ecgfrith East Anglia, Kent, Mercia
## 20                  Ecgfrith East Anglia, Kent, Mercia
## 22                   Cœnwulf East Anglia, Kent, Mercia
## 23       Cœnwulf and Cynehelm East Anglia, Kent, Mercia
## 24                   Cœnwulf East Anglia, Kent, Mercia
## 25                  Ceolwulf East Anglia, Kent, Mercia
## 26                 Beornwulf       East Anglia, Mercia
## 82      Ecgbehrt and Æthelwulf              Kent, Wessex
## 83      Ecgbehrt and Æthelwulf      Kent, Mercia, Wessex
## 84      Ecgbehrt and Æthelwulf              Kent, Wessex
## 85    Æthelwulf and Æðelstan I              Kent, Wessex
## 86                 Æthelwulf              Kent, Wessex
## 87 Æthelwulf and Æðelberht III              Kent, Wessex
## 88             Æðelberht III              Kent, Wessex
## 89                Æthelred I              Kent, Wessex
## 95                     Oswiu       Mercia, Northumbria
```

同样常见的是，王国的权力并非为某位统治者专有，而是由几个人分享（当一个强大的国王有好几个儿子时特别常见）。我们可以通过在 name 一栏中寻找逗号或“and”来发现这些例子。这一次，因为要同时寻找两件事情，所以更简单的做法是使用一个正则表达式而非固定的字符串。在正则表达式和 R 中，管道字符 | 的含义均为：或。

在下例中，为防止输出内容太多，我们只返回 name 一列，且忽略缺失值（使用 is.na）：

```
multiple_rulers <- str_detect(english_monarchs$name, ",|and")
english_monarchs$name[multiple_rulers & !is.na(multiple_rulers)]
```

```
##  [1] Sigeberht and Ecgric
##  [2] Hun, Beonna and Alberht
##  [3] Offa and Ecgfrith
##  [4] Cœnwulf and Cynehelm
##  [5] Sighere and Sebbi
##  [6] Sigeheard and Swaefred
##  [7] Eorcenberht and Eormenred
##  [8] Oswine, Swæfbehrt, Swæfheard
##  [9] Swæfbehrt, Swæfheard, Wihtred
## [10] Æðelberht II, Ælfric and Eadberht I
## [11] Æðelberht II and Eardwulf
## [12] Eadberht II, Eanmund and Sigered
## [13] Heaberht and Ecgbehrt II
## [14] Ecgbehrt and Æthelwulf
## [15] Ecgbehrt and Æthelwulf
## [16] Ecgbehrt and Æthelwulf
## [17] Æthelwulf and Æðelstan I
## [18] Æthelwulf and Æðelberht III
## [19] Penda and Eowa
## [20] Penda and Peada
## [21] Æthelred, Lord of the Mercians
## [22] Æthelflæd, Lady of the Mercians
## [23] Ælfwynn, Second Lady of the Mercians
## [24] Hálfdan and Eowils
## [25] Noðhelm and Watt
## [26] Noðhelm and Bryni
## [27] Noðhelm and Osric
## [28] Noðhelm and Æðelstan
## [29] Ælfwald, Oslac and Osmund
## [30] Ælfwald, Ealdwulf, Oslac and Osmund
## [31] Ælfwald, Ealdwulf, Oslac, Osmund and Oswald
## [32] Cenwalh and Seaxburh
## 211 Levels: Adda Æðelbehrt Æðelberht I ... Wulfhere
```

如果想把 name 一列拆分，使它列出每位统治者的名字，可以使用 str_split（或用 R 基本包中的 strsplit，作用基本一样）。str_split 接受一个向量作为输入参数，且将返回一个列表，这是因为每个输入的字符串可以被分成长度不同的向量。如果每个输入须返回相同的分割数，则可使用 str_split_fixed，它将返回一个矩阵。以下输出显示了多位统治者中的前几个例子：

```
individual_rulers <- str_split(english_monarchs$name, ", | and ")
head(individual_rulers[sapply(individual_rulers, length) > 1])
```

```
## [[1]]
## [1] "Sigeberht" "Ecgric"
##
## [[2]]
## [1] "Hun"     "Beonna"  "Alberht"
##
## [[3]]
## [1] "Offa"     "Ecgfrith"
##
```

```
## [[4]]
## [1] "Cœnwulf"  "Cynehelm"
##
## [[5]]
## [1] "Sighere" "Sebbi"
##
## [[6]]
## [1] "Sigeheard" "Swaefred"
```

在此期间，许多盎格鲁 – 撒克逊（Anglo-Saxon）统治者的名字中有古英语字符，像“æ”（“ash”），它代表“ae”或“ð”和“þ”（分别为“eth”和“thorn，”），它们都代表“th.”。在很多情况下，每个统治者名字的拼写并不一致，但要识别某位统治者，其名字拼写就必须固定下来。

让我们来看看有多少个 th、ð 和 þ 用来组成字母“th.”。我们可以用 str_count 计算出它们在每个名称中的出现次数，然后用 sum 来对所有统治者求和计算出总的出现次数：

```
th <- c("th", "ð", "þ")
sapply(           # 也可以使用 plyr 中的 laply
  th,
  function(th)
  {
    sum(str_count(english_monarchs$name, th))
  }
)

## th  ð  þ
## 74 26  7
```

在这个数据集开始看起来像常见的现代标准拉丁拼写法。如果要替换掉 eth 和 thorn 等字符串，可使用 str_replace_all。（有一个稍微不同的函数 str_replace，它仅仅替换掉第一个匹配的字符串。）把 eth 和 thorn 置于在方括号中的意思是：符合下列任一字符：

```
english_monarchs$new_name <- str_replace_all(english_monarchs$name, "[ðþ]", "th")
```

这种技巧对于清理一个类别变量的水平值非常有用。例如，性别在英语中可通过多种方式指定，但我们通常只需要其中的两个。在下例中，我们将匹配以“m”开头的（^）、后面跟着一个可选的（?）“ale”、且以字符串（$）为结尾：

```
gender <- c(
  "MALE", "Male", "male", "M", "FEMALE",
  "Female", "female", "f", NA
)
clean_gender <- str_replace(
  gender,
  ignore.case("^m(ale)?$"),
  "Male"
)
(clean_gender <- str_replace(
```

```
  clean_gender,
  ignore.case("^f(emale)?$"), "
  Female"
))

## [1] "Male" "Male" "Male" "Male" "Female" "Female" "Female" "Female"
## [9] NA
```

13.3　操作数据框

许多数据清理的任务涉及数据框的操作，要把它们转换成相应所需的形式。我们已知可用索引和 subset 函数来选择数据框的子集。其他常见任务，包括通过添加其他列（或替换现有列）来扩充数据框、处理缺失值，以及在数据框的宽和长格式之间的转换。以下有几个函数可用于在数据框中添加或替换列。

13.3.1　添加和替换列

假设要为 english_monarchs 数据框添加一列来表示统治者执政了多少年，我们可以使用标准的赋值操作符来完成：

```
english_monarchs$length.of.reign.years <-
  english_monarchs$end.of.reign - english_monarchs$start.of.reign
```

这是可行的，但要反复输入数据框变量名无论是对于输入或阅读来说都太麻烦。with 函数可以通过直接调用变量使之简化。它接受一个数据框 [2] 和要计算的表达式作为输入参数：

```
english_monarchs$length.of.reign.years <- with(
  english_monarchs,
  end.of.reign - start.of.reign
)
```

within 函数的工作方式类似，但它会返回整个数据框：

```
english_monarchs <- within(
  english_monarchs,
  {
    length.of.reign.years <- end.of.reign - start.of.reign
  }
)
```

在此例中 within 还需输入更多。当要更改多个列时，它更有用：

```
english_monarchs <- within(
  english_monarchs,
  {
```

注 2：或环境。

```
    length.of.reign.years <- end.of.reign - start.of.reign
    reign.was.more.than.30.years <- length.of.reign.years > 30
  }
)
```

也就是说，如果你正在创建或改变某一列，就使用 with；如果你想一次操作多个列，则用 within。

在 plyr 包中的 mutate 函数采取了另一种方法，它接受新的和更改的列，并把它们当成“名称－值”对[3]：

```
english_monarchs <- mutate(
  english_monarchs,
  length.of.reign.years        = end.of.reign - start.of.reign,
  reign.was.more.than.30.years = length.of.reign.years > 30
)
```

13.3.2　处理缺失值

在第 12 章，马鹿数据集显示了用四种不同方式测量得出每头鹿的颅容量。对其中某些而非所有的鹿，会进行第二次测量以测试该技术的可重复性。这意味着某些行会有缺失值。complete.cases 函数告诉我们哪些行没有缺失值：

```
data("deer_endocranial_volume", package = "learningr")
has_all_measurements <- complete.cases(deer_endocranial_volume)
deer_endocranial_volume[has_all_measurements, ]

##    SkullID VolCT VolBead VolLWH VolFinarelli VolCT2 VolBead2 VolLWH2
## 7     C120   346     335   1250          289    346      330    1264
## 8      C25   302     295   1011          250    303      295    1009
## 9       F7   379     360   1621          347    375      365    1647
## 10     B12   410     400   1740          387    413      395    1728
## 11     B17   405     395   1652          356    408      395    1639
## 12     B18   391     370   1835          419    394      375    1825
## 13      J7   416     405   1834          408    417      405    1876
## 15      A4   336     330   1224          283    345      330    1192
## 20      K2   349     355   1239          286    354      365    1243
```

捷径为 na.omit 函数，它能删除数据框中所有带有缺失值的行[4]：

```
na.omit(deer_endocranial_volume)

##    SkullID VolCT VolBead VolLWH VolFinarelli VolCT2 VolBead2 VolLWH2
## 7     C120   346     335   1250          289    346      330    1264
## 8      C25   302     295   1011          250    303      295    1009
## 9       F7   379     360   1621          347    375      365    1647
## 10     B12   410     400   1740          387    413      395    1728
```

注 3：R 基本包中的 transform 函数是 mutate 函数的早期实现，现已过时。

注 4：na.exclude 与 na.omit 的行为一样，两者同存为惯例。

```
## 11     B17   405     395   1652          356    408      395    1639
## 12     B18   391     370   1835          419    394      375    1825
## 13      J7   416     405   1834          408    417      405    1876
## 15      A4   336     330   1224          283    345      330    1192
## 20      K2   349     355   1239          286    354      365    1243
```

相反地，如果你的数据框中包含任何缺失值，na.fail 将抛出一个错误：

```
na.fail(deer_endocranial_volume)
```

这两个函数也可以对向量进行缺失值删除或者抛出错误，就像在数据框中一样。

> 可以采取统计上可行的多次插补来填补缺失值。这已超出本书的范围，不过我们建议你从 mice 和 mix 包开始研究。

13.3.3 在宽和长表格之间进行转换

马鹿数据集包含了以四种不同方式获得的鹿头骨颅容量数据。它的每一列包含了对某种类型的鹿的测量结果。（为简单起见，我们忽略了重复测量的列。）这就是数据框的宽形式：

```
deer_wide <- deer_endocranial_volume[, 1:5]
```

从另一个角度看，每一列中的头骨测量数据都是同一类型的东西（即测量值），只是测量的方式不同。因此，另一种表示该数据的方式是：每个鹿都有 4 行数据，每行有以下几列：一列是和之前一样颅骨的 ID（所以每个值将被重复四次），一列为测量值，还有一列用于解释本行所在的测量类型的因子。这就是所谓的数据框的长格式。

R 的基础包中 reshape 的函数能用于宽和长形式之间的转换。它的功能非常强大，但并不太直观。更好的选择是使用 reshape2 包中的功能。

在这个包中可用 melt 函数将宽形式转换成长形式[5]。我们选择 SkullID 作为 ID 列（与其他所有都被归为一次测量）：

```
library(reshape2)
deer_long <- melt(deer_wide, id.vars = "SkullID")
head(deer_long)

##   SkullID variable value
## 1   DIC44    VolCT   389
## 2     B11    VolCT   389
## 3   DIC90    VolCT   352
## 4   DIC83    VolCT   388
## 5  DIC787    VolCT   375
## 6 DIC1573    VolCT   325
```

注 5：术语“melting”及其反义词“casting”均参考自钢铁行业。

或者，使用 measure.vars 参数，它显示除了 id.vars 列之外的其他所有列。在这种情况下，工作量会变大，但当有很多的 ID 变量和极少测量变量时，它很有用：

```
melt(deer_wide, measure.vars = c("VolCT", "VolBead", "VolLWH", "VolFinarelli"))
```

dcast 函数能把长转换回宽形式，并将结果返回为一个数据框（相关函数 acast 返回一个向量、矩阵或数组）：

```
deer_wide_again <- dcast(deer_long, SkullID ~ variable)
```

我们重构的数据集 deer_wide_again 与原来的 deer_wide 基本一样，除了它按 SkullID 中的字母顺序排序。

电子表格的爱好者可能会注意到，acast 和 dcast 能有效地创建数据透视表。

13.3.4 使用SQL

sqldf 包提供了使用 SQL 操作数据框的方式。一般情况下，标准 R 里面的函数都比 SQL 代码要更简洁、可读性也更好。不过，当你原来的工作背景是数据库，这个包能帮你平滑过渡到 R：

```
install.packages("sqldf")
```

下例将比较标准 R 和 sqldf 版本的子集查询语句：

```
library(sqldf)

## Loading required package: DBI

## Loading required package: gsubfn

## Loading required package: proto

## Loading required namespace: tcltk

## Loading required package: chron

## Loading required package: RSQLite

## Loading required package: RSQLite.extfuns

subset(
  deer_endocranial_volume,
  VolCT > 400 | VolCT2 > 400,
  c(VolCT, VolCT2)
)
```

```
##    VolCT VolCT2
## 10   410    413
## 11   405    408
## 13   416    417
## 16   418     NA
```

```
query <-
  "SELECT
    VolCT,
    VolCT2
  FROM
    deer_endocranial_volume
  WHERE
    VolCT > 400 OR
    VolCT2 > 400"
sqldf(query)
```

```
## Loading required package: tcltk
```

```
##   VolCT VolCT2
## 1   410    413
## 2   405    408
## 3   416    417
## 4   418     NA
```

13.4 排序

把数值数据按大小进行排序通常很有用，因为我们关心的值往往就在两端。sort 函数按向量从小到大（或从大到小）排序[6]：

```
x <- c(2, 32, 4, 16, 8)
sort(x)
```

```
## [1]  2  4  8 16 32
```

```
sort(x, decreasing = TRUE)
```

```
## [1] 32 16  8  4  2
```

字符串也可以排序，但其顺序往往取决于语言环境。通常字母从 a 到 z 排序，但也有些比较奇怪的顺序：例如在爱沙尼亚语中，z 位于 s 之后、t 之前。其他奇怪的顺序都列在 ?Comparision 帮助页面中。在英语或北美地区，你会看到这样的结果：

```
sort(c("I", "shot", "the", "city", "sheriff"))
```

```
## [1] "city"    "I"       "sheriff" "shot"    "the"
```

注 6：痴迷于排序算法的高手可能会指出 sort 函数默认使用 shellsort，但你亦可指定 method = quick 而使用快速排序。基数排序也可用于因子中。

order 函数是 sort 的某种逆转操作。order 中的第 i 个元素是 x 中的元素在排序之后最终将出现的位置。这种说法有点令人头晕，你需要了解的是：x[order(x)] 将返回与 sort(x) 相同的结果：

```
order(x)

## [1] 1 3 5 4 2

x[order(x)]

## [1]  2  4  8 16 32

identical(sort(x), x[order(x)])

## [1] TRUE
```

order 在对数据框排序时非常有用，因为数据框不能直接使用 sort。例如，使 english_monarchs 数据框按统治的年份开始排序，可用：

```
year_order <- order(english_monarchs$start.of.reign)
english_monarchs[year_order, ]
```

plyr 包中的 arrange 函数提供了一个替代函数，它只用一行就能对数据框排序：

```
arrange(english_monarchs, start.of.reign)
```

rank 函数为数据集中的每个元素给出了排名，提供了排名情况相等时的几种处理方法：

```
(x <- sample(3, 7, replace = TRUE))

## [1] 1 2 1 3 3 3 2

rank(x)

## [1] 1.5 3.5 1.5 6.0 6.0 6.0 3.5

rank(x, ties.method = "first")

## [1] 1 3 2 5 6 7 4
```

13.5 函数式编程

如 LISP 和 Haskell 的函数式编程语言的概念已被引入到 R 中。你无须了解任何函数式编程就可以使用它们[7]。你只需要知道，这些功能在数据操作时很有用。

Negate 函数接受一个谓词（predicate，即一个返回逻辑向量的函数），并返回另一个刚好相

注 7：如有兴趣可以参考 wordIQ.com 上的一个很好的介绍。

反的谓词[8]。当输入是 FLASE 时，它将返回 TRUE；输入是 FALSE，它返回 TRUE：

```
ct2 <- deer_endocranial_volume$VolCT2   # 为了方便输入
isnt.na <- Negate(is.na)
identical(isnt.na(ct2), !is.na(ct2))
```

```
## [1] TRUE
```

Filter 可以接受两个参数：第一个参数为返回一个逻辑向量的函数；其次为一个输入向量，它只有在函数返回为 TRUE 时才返回那些值：

```
Filter(isnt.na, ct2)
```

```
## [1] 346 303 375 413 408 394 417 345 354
```

Position 函数的行为有点像第 4.2 节的 which 函数。它将返回第一个把谓词应用于矢量上而返回 TRUE 的索引下标：

```
Position(isnt.na, ct2)
```

```
## [1] 7
```

Find 与 Position 类似，但它返回的是第一个值而不是第一个索引：

```
Find(isnt.na, ct2)
```

```
## [1] 346
```

Map 函数将一个函数应用于输入参数中的每个元素上。它只是使用了 SIMPLIFY = FALSE 选项的 mapply 函数的封装函数。在下例中，我们将使用以上各种方法取得马鹿数据集中每头鹿测量的平均值。首先，我们需要一个用于传递给 Map 的函数，它能查找出每一个鹿头骨的体积：

```
get_volume <- function(ct, bead, lwh, finarelli, ct2, bead2, lwh2)
{
  # 如果有第二次测量，取平均值
  if(!is.na(ct2))
  {
    ct <- (ct + ct2) / 2
    bead <- (bead + bead2) / 2
    lwh <- (lwh + lwh2) / 2
  }
  # 把 lwh 除以 4，使它与其他测量结果看齐
  c(ct = ct, bead = bead, lwh.4 = lwh / 4, finarelli = finarelli)
}
```

注 8：从技术上讲，谓词 predicate 是一个返回单个逻辑值的函数，我们在这里滥用了这个词，尽管这个词的矢量版本还没有被创造出来。我比较喜欢施瓦辛格式的 predicator，但在它流行之前还是沿用 predicate 吧。

然后，Map 的行为就和 mapply 一样：它接收一个函数作为第一个参数，然后其他所有的参数会逐个传递给它：

```
measurements_by_deer <- with(
  deer_endocranial_volume,
  Map(
    get_volume,
    VolCT,
    VolBead,
    VolLWH,
    VolFinarelli,
    VolCT2,
    VolBead2,
    VolLWH2
  )
)
head(measurements_by_deer)

## [[1]]
##        ct      bead     lwh.4 finarelli
##       389       375       371       337
##
## [[2]]
##        ct      bead     lwh.4 finarelli
##     389.0     370.0     430.5     377.0
##
## [[3]]
##        ct      bead     lwh.4 finarelli
##     352.0     345.0     373.8     328.0
##
## [[4]]
##        ct      bead     lwh.4 finarelli
##     388.0     370.0     420.8     377.0
##
## [[5]]
##        ct      bead     lwh.4 finarelli
##     375.0     355.0     364.5     328.0
##
## [[6]]
##        ct      bead     lwh.4 finarelli
##     325.0     320.0     340.8     291.0
```

Reduce 函数能把一个二元函数转变为一个接受多个输入的函数。例如，+ 运算符能计算两个数字的总和，但 sum 函数能计算出多个输入的总和。sum(a, b, c, d, e)（大致）相当于 Reduce("+", list(a, b, c, d, e))。

我们可以定义一个简单的二元函数来（并行地）计算出两个输入值中的最大值：

```
pmax2 <- function(x, y) ifelse(x >= y, x, y)
```

如果对这个函数进行 reduce 操作，那么它能接受一系列的输入（正如在 R 基础包中的 pmax 函数一样）：

```
Reduce(pmax2, measurements_by_deer)

##        ct      bead     lwh.4 finarelli
##    418.00    405.00    463.75    419.00
```

Reduce 的一个限制条件是，它将对其输入成对地反复调用二元函数，也就是说：

```
Reduce("+", list(a, b, c, d, e))
```

下面的用法也是一样的：

```
((((a + b) + c) + d) + e)
```

这意味着，它不能用于例如求平均值这样的计算，因为：

```
mean(mean(mean(mean(a, b), c), d), e) != mean(a, b, c, d, e)
```

13.6　小结

- stringr 包对操作字符串非常有用。
- 数据框的列可以被相加、相减或操作。
- 数据框能以宽或长的形式存在。
- 可以对向量进行排序、排名和 order。
- R 有一些函数式编程的能力，如 Map 和 Reduce。

13.7　知识测试：问题

- 问题 13-1
  如何计算单词“thou”出现在莎士比亚的《暴风雨》中出现的次数？

- 问题 13-2
  想一想如何能为数据框添加列，指出尽可能多的函数名。

- 问题 13-3
  “melting”相反的用法是什么？

- 问题 13-4
  如何对数据框按列重新排序？

- 问题 13-5
  如何找到向量中的第一个正数？

13.8 知识测试：练习

- 练习 13-1

  从 `learningr` 包中加载 `hafu` 数据集。在 `Father` 和 `Mother` 列中，有一些值在国家名后面带有问号，表明作者不确定这些父母的国籍。在 `fafu` 数据框中创建两个新的列，分别表示是否在 `Father` 或 `Mother` 列中带有问号。

  从 `Father` 和 `Mother` 列中删除那些问号。[10]

- 练习 13-2

  `hafu` 数据集中每个父母的国籍都有单独的列。把数据框从宽形转换为长形，使一列为父母的国籍，一列为国籍所对应的父母是谁。[5]

- 练习 13-3

  写一个函数，它能返回向量中 10 个最常见的值及其次数。尝试把这个函数应用于 `hafu` 数据集的某些列中。[10]

第 14 章

探索和可视化

当导入数据、完成清理工作，并把它们转换成某种合适的形式后，你会问“所有这些都意味着什么？”你可以使用的两个主要工具是汇总统计和绘图（建模是后话，因为你首先需要理解数据，然后才能对它们正确地建模）。对于计算统计任务来说，R 有一系列非常全面的函数以及三种不同的绘图系统可供选择。

14.1　本章目标

阅读完本章后，你会了解以下内容：

- 如何对一定范围内的数值数据进行汇总统计计算；
- 如何使用 R 的三种绘图系统绘制标准图形；
- 如何简单地操作这些绘图方法。

14.2　汇总统计

我们已经了解了很多用于计算汇总统计的函数，本节算是一个小结。大多数函数的命名及其用法都显而易见，例如，`mean` 和 `median` 函数的计算任务和它们的名字相同。没有求模的函数，但可以从 `table` 函数的结果计算出来，它给出了每个元素的计数结果。（如果你了解得还不够，现在再去做一下练习 13-3 吧。）

在下例中，`obama_vs_mccain` 数据集包含在 2008 年美国总统选举中投票给奥巴马和麦凯恩的人口组成数据，以及一些相关的人口统计信息：

```
data(obama_vs_mccain, package = "learningr")
obama <- obama_vs_mccain$Obama
mean(obama)

## [1] 51.29

median(obama)

## [1] 51.38
```

table 函数对于 Obama 变量（或其他的数值变量）来说意义不是很大，因为每个值都是唯一的。通过与 cut 相结合，我们可以看见有多少值落在了不同的分组中：

```
table(cut(obama, seq.int(0, 100, 10)))

##
##  (0,10]  (10,20]  (20,30]  (30,40]  (40,50]  (50,60]  (60,70]  (70,80]
##       0        0        0        8       16       16        9        1
## (80,90] (90,100]
##       0        1
```

var 和 sd 分别计算方差和标准差。用于计算平均绝对偏差的 mad 函数则比较少见：

```
var(obama)

## [1] 123.1

sd(obama)

## [1] 11.09

mad(obama)

## [1] 11.49
```

有几个函数可用于获取数值数据的极值。例如，最常见的 min 和 max，它们能分别给出输入的最小值和最大值。pmin 和 pmax（“并行”版本）则能计算相同长度的若干向量中在相同位置的最小和最大值。而 range 函数只需一行函数就能给出最小和最大值：

```
min(obama)

## [1] 32.54

with(obama_vs_mccain, pmin(Obama, McCain))

##  [1] 38.74 37.89 44.91 38.86 36.91 44.71 38.22 6.53 36.93 48.10 46.90
## [12] 26.58 35.91 36.74 48.82 44.39 41.55 41.15 39.93 40.38 36.47 35.99
## [23] 40.89 43.82 43.00 49.23 47.11 41.60 42.65 44.52 41.61 41.78 36.03
## [34] 49.38 44.50 46.80 34.35 40.40 44.15 35.06 44.90 44.75 41.79 43.63
## [45] 34.22 30.45 46.33 40.26 42.51 42.31 32.54

range(obama)

## [1] 32.54 92.46
```

cumin 和 cummax 能分别计算向量中的最小值和最大值。同样地，cumsum 和 cumprod 能计算数据中的累加与累乘。当输入的数据是有序的，这些函数会非常有用：

```
cummin(obama)

##  [1] 38.74 37.89 37.89 37.89 37.89 37.89 37.89 37.89 37.89 37.89 37.89
## [12] 37.89 35.91 35.91 35.91 35.91 35.91 35.91 35.91 35.91 35.91 35.91
## [23] 35.91 35.91 35.91 35.91 35.91 35.91 35.91 35.91 35.91 35.91 35.91
## [34] 35.91 35.91 35.91 34.35 34.35 34.35 34.35 34.35 34.35 34.35 34.35
## [45] 34.22 34.22 34.22 34.22 34.22 34.22 32.54

cumsum(obama)

##  [1]   38.74   76.63  121.54  160.40  221.34  275.00  335.59  428.05
##  [9]  489.96  540.87  587.77  659.62  695.53  757.38  807.23  861.16
## [17]  902.71  943.86  983.79 1041.50 1103.42 1165.22 1222.55 1276.61
## [25] 1319.61 1368.84 1415.95 1457.55 1512.70 1566.83 1623.97 1680.88
## [33] 1743.76 1793.46 1837.96 1889.34 1923.69 1980.44 2034.91 2097.77
## [41] 2142.67 2187.42 2229.21 2272.84 2307.06 2374.52 2427.15 2484.49
## [49] 2527.00 2583.22 2615.76

cumprod(obama)

##  [1] 3.874e+01 1.468e+03 6.592e+04 2.562e+06 1.561e+08 8.377e+09 5.076e+11
##  [8] 4.693e+13 2.905e+15 1.479e+17 6.937e+18 4.984e+20 1.790e+22 1.107e+24
## [15] 5.519e+25 2.976e+27 1.237e+29 5.089e+30 2.032e+32 1.173e+34 7.261e+35
## [22] 4.487e+37 2.572e+39 1.391e+41 5.980e+42 2.944e+44 1.387e+46 5.769e+47
## [29] 3.182e+49 1.722e+51 9.841e+52 5.601e+54 3.522e+56 1.750e+58 7.789e+59
## [36] 4.002e+61 1.375e+63 7.801e+64 4.249e+66 2.671e+68 1.199e+70 5.367e+71
## [43] 2.243e+73 9.785e+74 3.349e+76 2.259e+78 1.189e+80 6.817e+81 2.898e+83
## [50] 1.629e+85 5.302e+86
```

可能如你所料，quantile 函数能够计算分位数（median、min 和 max 是特殊情况）。它默认计算中位数、最小值、最大值和上下四分位数，而且它令人惊叹地进行了过度设计，竟然提供了 9 种不同的算法：

```
quantile(obama)

##    0%   25%   50%   75%  100%
## 32.54 42.75 51.38 57.34 92.46

quantile(obama, type = 5)  # 重现 SAS 的结果

##    0%   25%   50%   75%  100%
## 32.54 42.63 51.38 57.34 92.46

quantile(obama, c(0.9, 0.95, 0.99))

##   90%   95%   99%
## 61.92 65.17 82.16
```

IQR 封装了 quantile 函数，它能直接计算出四分位差（第 75 百分位减去第 25 个百分位）：

```
IQR(obama)
```

```
## [1] 14.58
```

fivenum 对 quantile 作了大量简化而且运行更快。你只能使用一种算法，并且只能计算出默认的位数。它很适用于对速度要求很高的那些场景：

```
fivenum(obama)
```

```
## [1] 32.54 42.75 51.38 57.34 92.46
```

有些快捷方式能够一次完成多项统计计算。例如，你已见过的能接受向量或数据框的 summary 函数：

```
summary(obama_vs_mccain)
```

```
##          State          Region         Obama            McCain
##  Alabama    : 1   IV      : 8   Min.   :32.5   Min.   : 6.53
##  Alaska     : 1   I       : 6   1st Qu.:42.8   1st Qu.:40.39
##  Arizona    : 1   III     : 6   Median :51.4   Median :46.80
##  Arkansas   : 1   V       : 6   Mean   :51.3   Mean   :47.00
##  California: 1   VIII    : 6   3rd Qu.:57.3   3rd Qu.:55.88
##  Colorado   : 1   VI      : 5   Max.   :92.5   Max.   :65.65
##  (Other)    :45   (Other):14
##     Turnout       Unemployment       Income         Population
##  Min.   :50.8   Min.   :3.40   Min.   :19534   Min.   :  563626
##  1st Qu.:61.0   1st Qu.:5.05   1st Qu.:23501   1st Qu.: 1702662
##  Median :64.9   Median :5.90   Median :25203   Median : 4350606
##  Mean   :64.1   Mean   :6.01   Mean   :26580   Mean   : 6074128
##  3rd Qu.:68.0   3rd Qu.:7.25   3rd Qu.:28978   3rd Qu.: 6656506
##  Max.   :78.0   Max.   :9.40   Max.   :40846   Max.   :37341989
##  NA's   :4
##     Catholic       Protestant        Other       Non.religious     Black
##  Min.   : 6.0   Min.   :26.0   Min.   :0.00   Min.   : 5    Min.   : 0.4
##  1st Qu.:12.0   1st Qu.:46.0   1st Qu.:2.00   1st Qu.:12    1st Qu.: 3.1
##  Median :21.0   Median :54.0   Median :3.00   Median :15    Median : 7.4
##  Mean   :21.7   Mean   :53.8   Mean   :3.29   Mean   :16    Mean   :11.1
##  3rd Qu.:29.0   3rd Qu.:62.0   3rd Qu.:4.00   3rd Qu.:19    3rd Qu.:15.2
##  Max.   :46.0   Max.   :80.0   Max.   :8.00   Max.   :34    Max.   :50.7
##  NA's   :2      NA's   :2      NA's   :2      NA's   :2
##      Latino       Urbanization
##  Min.   : 1.2   Min.   :   1
##  1st Qu.: 4.3   1st Qu.:  46
##  Median : 8.2   Median : 101
##  Mean   :10.3   Mean   : 386
##  3rd Qu.:12.1   3rd Qu.: 221
##  Max.   :46.3   Max.   :9856
##
```

cor 函数能计算数值向量之间的相关性。你可能会想到，投票给奥巴马和投票给麦凯恩的选民之间应该会有一个近乎完美的负相关关系（其中的瑕疵是由独立候选人的选民造成的）。cancor 函数（典型相关“canonical correlation”的简称）能提供更多的细节，而 cov

函数能计算协方差：

```
with(obama_vs_mccain, cor(Obama, McCain))

## [1] -0.9981

with(obama_vs_mccain, cancor(Obama, McCain))

## $cor
## [1] 0.9981
##
## $xcoef
##         [,1]
## [1,] 0.01275
##
## $ycoef
##          [,1]
## [1,] -0.01287
##
## $xcenter
## [1] 51.29
##
## $ycenter
## [1] 47

with(obama_vs_mccain, cov(Obama, McCain))

## [1] -121.7
```

14.3 三种绘图系统

R 在其生命周期中已经累计开发了三种不同的图形系统。`base` 图形是最古老的系统，在 R 的存在初期它就已经存在了。`base` 图形很容易上手，不过它们需要大量的修改甚至一些魔法咒语才能变得更完美，而且它很难扩展到新的图表类型中。

`grid` 图形系统的开发旨在修正 `base` 中的一些限制而使绘图更加灵活。`grid` 允许你在绘图时涉及系统底层，可以具体指定在哪里画每个点、线或矩形。虽然这听起来不错，但当我们想画散点图时，没有人想每次都花大量的时间写几百行代码来实现。

第二个图形系统 `lattice` 建立在 `grid` 系统之上，它为所有常见的图表类型提供了高级函数。它有两个突出的特点是 `base` 图形系统所不具备的。首先，每个绘图的结果被能保存到一个变量中，而不仅仅是绘制在屏幕上。这意味着你可以先画一些图形，再对其进行编辑，然后再把它画出来；这样会更容易画出那些相关的图形，而且图形可以保存在会话（session）之间。第二大特点是，可在一个格子中包含多个面板 [1]，因此你能把数据分成不

注 1：它有好几种术语名称。Edward Tufte 在 *Envisioning Information* 中称之为“small multiples”，贝尔实验室的 Bill Cleveland 和 Rick Becker 为它杜撰了“trellising”一词，Deepayan Sarkar 在 `lattice` 包中为了避免使用以上的名称，称之为“latticing”，而 Leland Wilkingson 把它称为“faceting”（`ggplot2` 中的术语）。

同的类别并比较各组之间的差异。这就解决了我们在第 9 章讨论的拆分－应用－合并所对应的绘图问题。

同样也是建立在 `grid` 系统之上的 `ggplot2` 系统是三个图形系统中最新的。gg 代表：grammar of graphics（制图语法）[2]，其目标在于把图形分解成不同的组块。其结果就是，ggplot 的代码看起来有点像是用英文描述你在图表中做什么。

令人遗憾的是，这三个系统几乎互不兼容（也有一些方法能把 `base` 和 `grid` 系统组合起来，然而万不得已最好不要使用它）。好消息是，你几乎可以使用 `ggplot2` 做任何事情，所以没有必要对所有三个系统都学习一遍。但在极少的某些场景下，ggplot2 不太适用——与其他图形系统相比它需要更多的计算，因此，如果想快速和粗略地绘制非常大的数据集，使用其他系统会更方便。此外，许多图形软件包是基于另外两个系统的，所以如果想使用那些包，就需要对 `base` 或 `lattice` 系统稍作了解。

下例展示了所有的三个系统。如果时间紧迫，你可以仅需阅读 `ggplot2` 部分。由于篇幅的限制，本章只能浅尝辄止而不作深入探讨。幸好，关于 R 的图形绘制，有三本非常优秀和通俗易懂的书：*R Graphics*、*ggplot2* 和 *Lattice*，其作者分别是 `grid`、`ggplot2` 和 `lattice` 系统的创作者[3]。

14.4 散点图

也许在所有图形中最常见就是散点图，它用于研究两个连续变量之间的关系。`obama_vs_mccain` 数据集中有很多可比较的数值变量，不过我们先考虑这个问题：“选民的收入是否会影响投票率？”

14.4.1 第一种方法：base绘图法

`base` 中用于绘制散点图的函数就是简单的 `plot`。最近流行的编码风格最佳实践就是把所有你想要用于绘图的变量置于一个（或数个）数据框中，而不是把它们分置于几个单独的向量中。不过遗憾的是，`plot` 比这种想法出现得早[4]，所以我们必须把它包括在 `with` 的函数调用中来访问列。

尽管 `plot` 会简单地忽略缺失值，但是为了代码的整洁，删除那些没有 `Turnout` 值的行吧：

```
obama_vs_mccain <- obama_vs_mccain[!is.na(obama_vs_mccain$Turnout), ]
```

注 2：此概念由 Leland Wilkinson 在同名书中提出。这本书很不错，但由于到处都是公式，不适合床上阅读。
注 3：绘图方面的书的好处是：即使你不太有兴趣，快速翻阅这些书也挺赏心悦目。
注 4：predate 这里指的是“出现得早”，而不是“捕获并吃掉”。

然后，我们就可以创建一个简单的散点图，如图 14-1 所示。

```
with(obama_vs_mccain, plot(Income, Turnout))
```

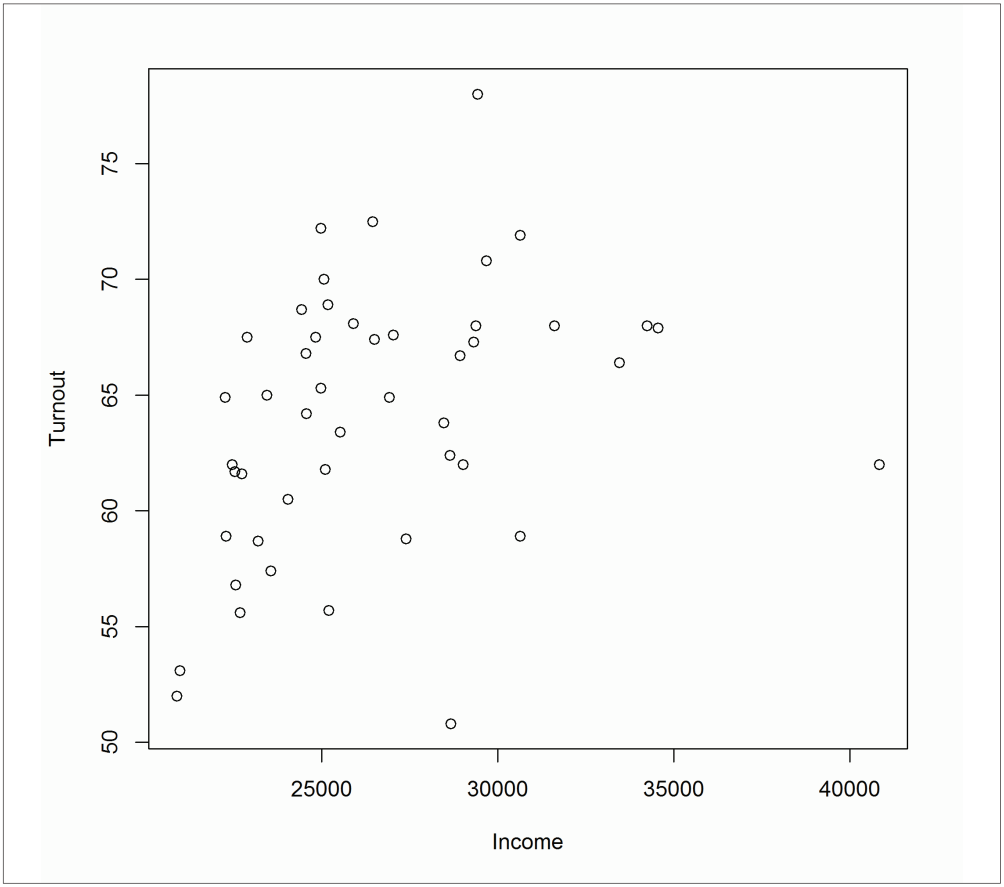


图 14-1：使用 base 图形系统画的简单散点图

plot 有很多参数可以用于自定义输出格式，其中一些比其他的更直观。col 能改变点的颜色。它可以接受任何通过 colors 返回的已命名的颜色，或者像 "#1234556" 的 HTML 风格的十六进制值。你可使用 pch（即 plot character 绘图字符的缩写）来改变点的形状[5]。图 14-2 显示了更新后的散点图，它把颜色变成紫色，并使用点状来填充圈点。

```
with(obama_vs_mccain, plot(Income, Turnout, col = "violet", pch = 20))
```

注 5：阅读 ?points 的帮助页面，然后试试 plot(1:25, pch = 1:25, bg = "blue") 来看看不同的形状。

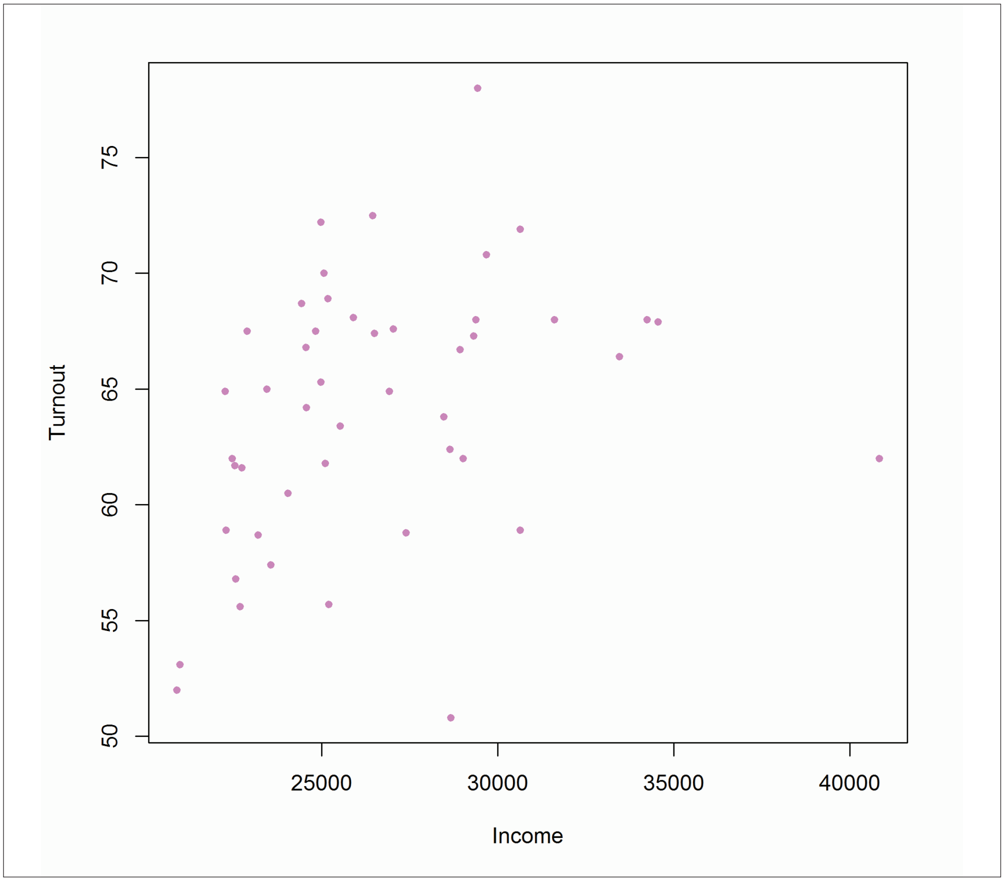


图 14-2：使用 base 图形系统设置颜色和点的形状

可通过 log 参数来设置对数坐标。log = "x" 表示使用 x 轴为对数坐标，log = "y" 表示使用 y 轴为对数坐标，而 log = "xy" 则表示同时使用 x 和 y 轴作为对数坐标。图 14-3 和图 14-4 显示了对数轴坐标的一些选项：

```
with(obama_vs_mccain, plot(Income, Turnout, log = "y"))
# 图 14-3

with(obama_vs_mccain, plot(Income, Turnout, log = "xy"))
# 图 14-4
```

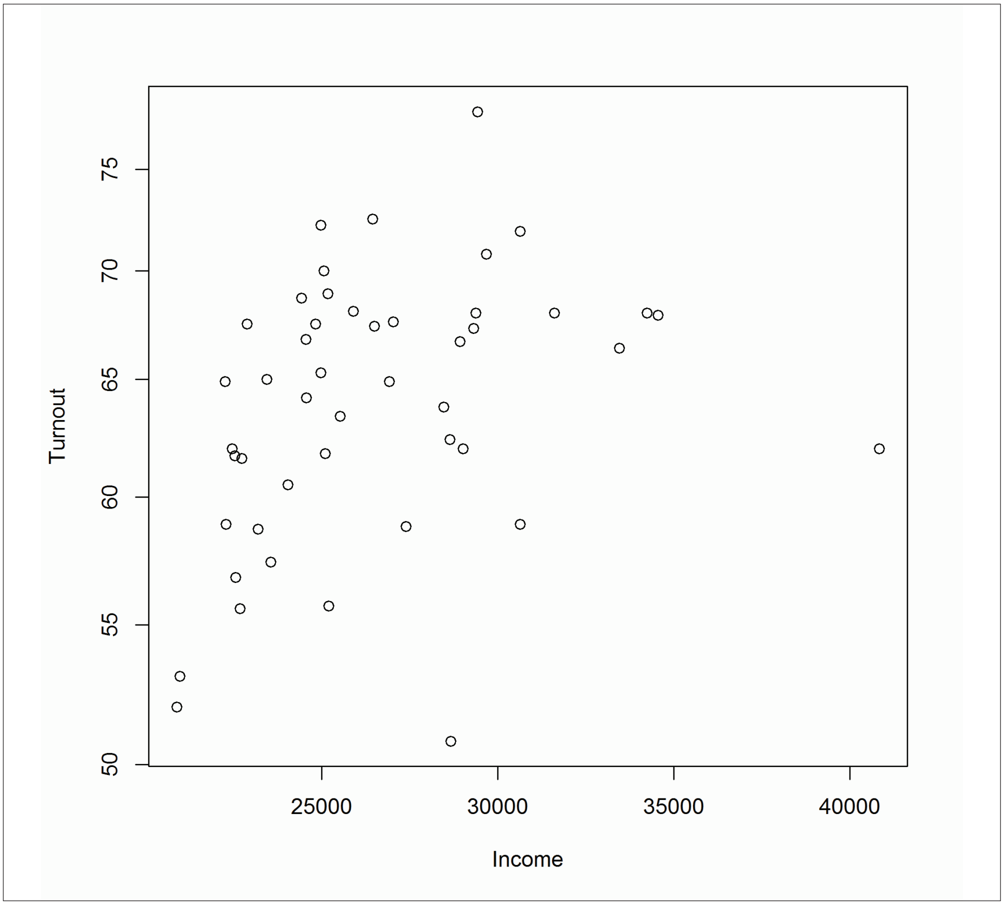


图 14-3：使用 base 图形系统的 y 轴对数坐标

我们可以看到，收入和投票率之间有明显的正相关关系，并且在对数坐标中这种相关性更强。下一个问题是："这种关系在美国所有地区中是否都一样？"要回答这个问题，可根据 `Region` 列把数据分割成最多 10 个标准联邦区，并把"矩阵"中的每个子集都绘制到一张图上。`layout` 函数用来控制矩阵中多个绘图区的布局。对于以下代码段，没必须花费大多的时间去理解它的意思，它只是用来证明使用 `base` 图形系统把多个相关的图形绘制到一起是可能的。不过遗憾的是，这段代码看起来就像从传说中的丑树中跌下来一样难看，所以这种技术还是少用为好。图 14-5 显示了其结果：

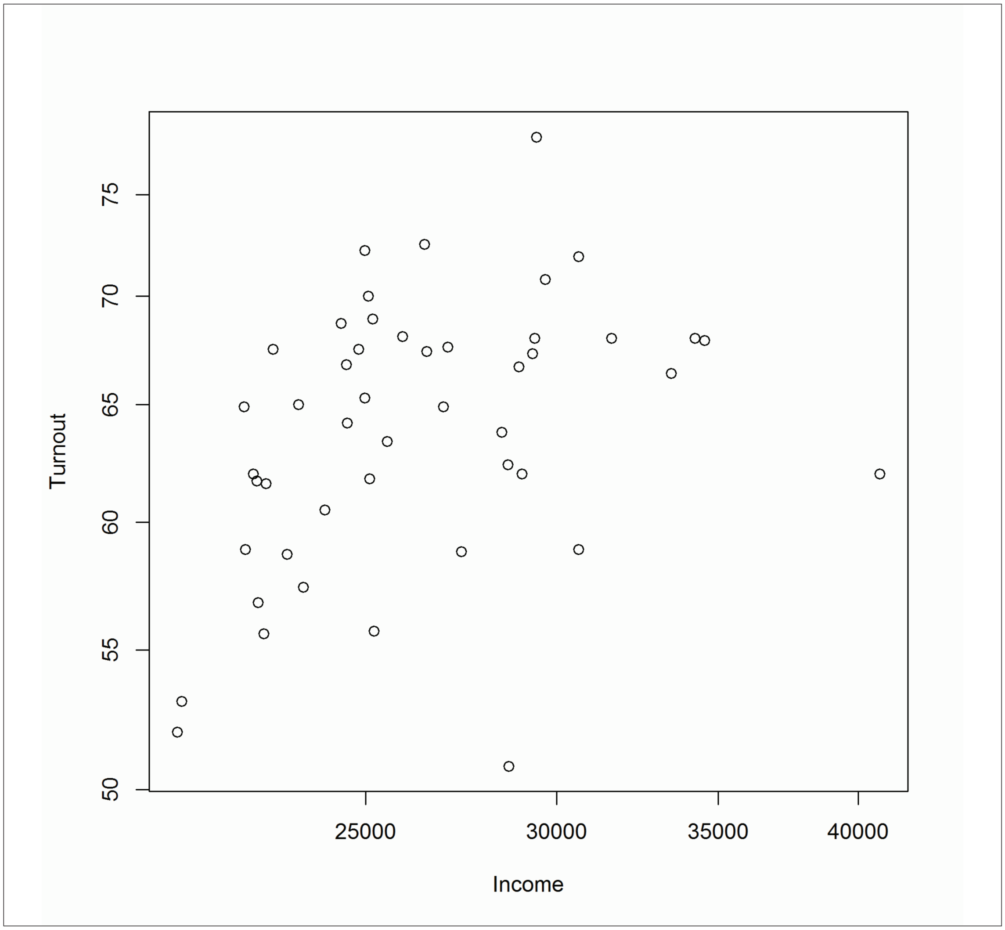


图 14-4：使用 base 图形系统的 x 和 y 轴对数坐标

```
par(mar = c(3, 3, 0.5, 0.5), oma = rep.int(0, 4), mgp = c(2, 1, 0))
regions <- levels(obama_vs_mccain$Region)
plot_numbers <- seq_along(regions)
layout(matrix(plot_numbers, ncol = 5, byrow = TRUE))
for(region in regions)
{
  regional_data <- subset(obama_vs_mccain, Region == region)
  with(regional_data, plot(Income, Turnout))
}
```

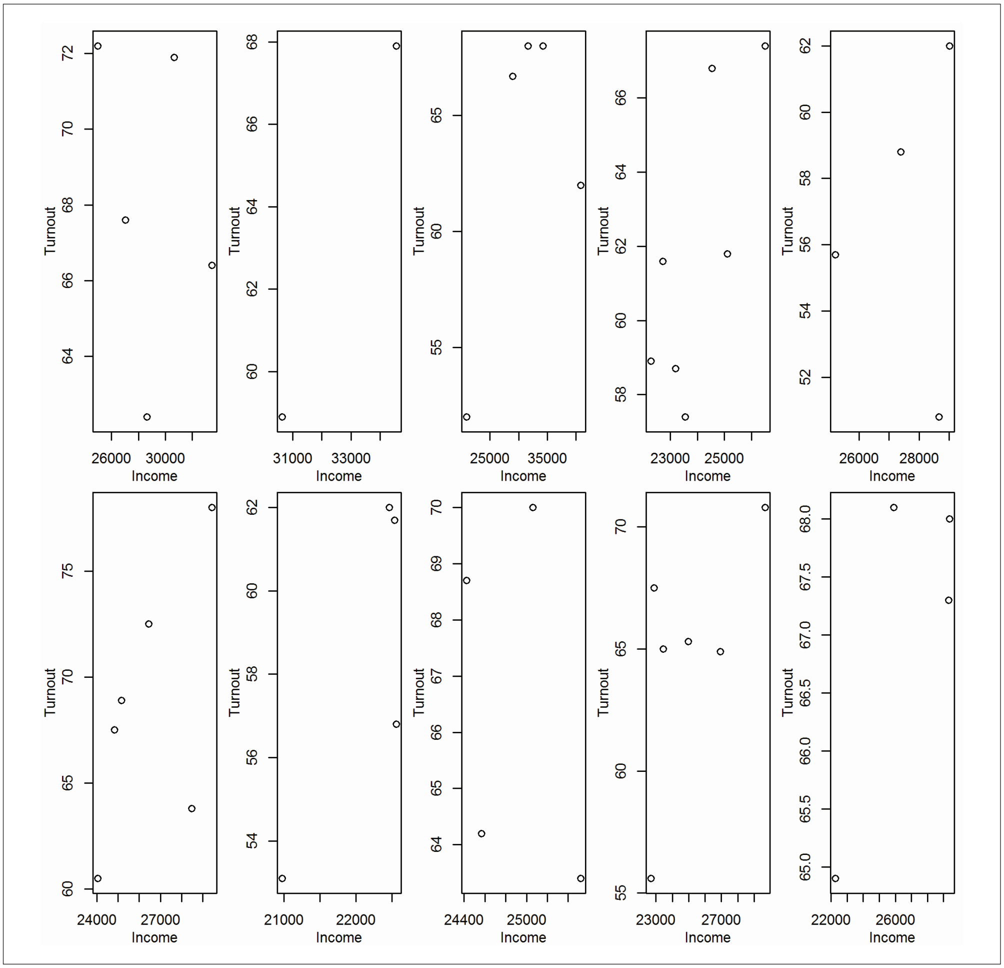


图 14-5：使用 base 图形系统在同一张图中显示多个子图

14.4.2 第二种方法：lattice图形系统

lattice 版本的 plot 是 xyplot。它使用了一个公式接口来指定 x 和 y 坐标变量。公式将在 15.4 节中深入讨论，现在你需要做的是输入 yvar ~ xvar。接着，xyplot（和其他 lattice 函数）还需要一个 data 参数，此参数将告诉它从哪个数据框中寻找变量。图 14-6 是图 14-1 的 lattice 版本：

```
library(lattice)
xyplot(Turnout ~ Income, obama_vs_mccain)
```

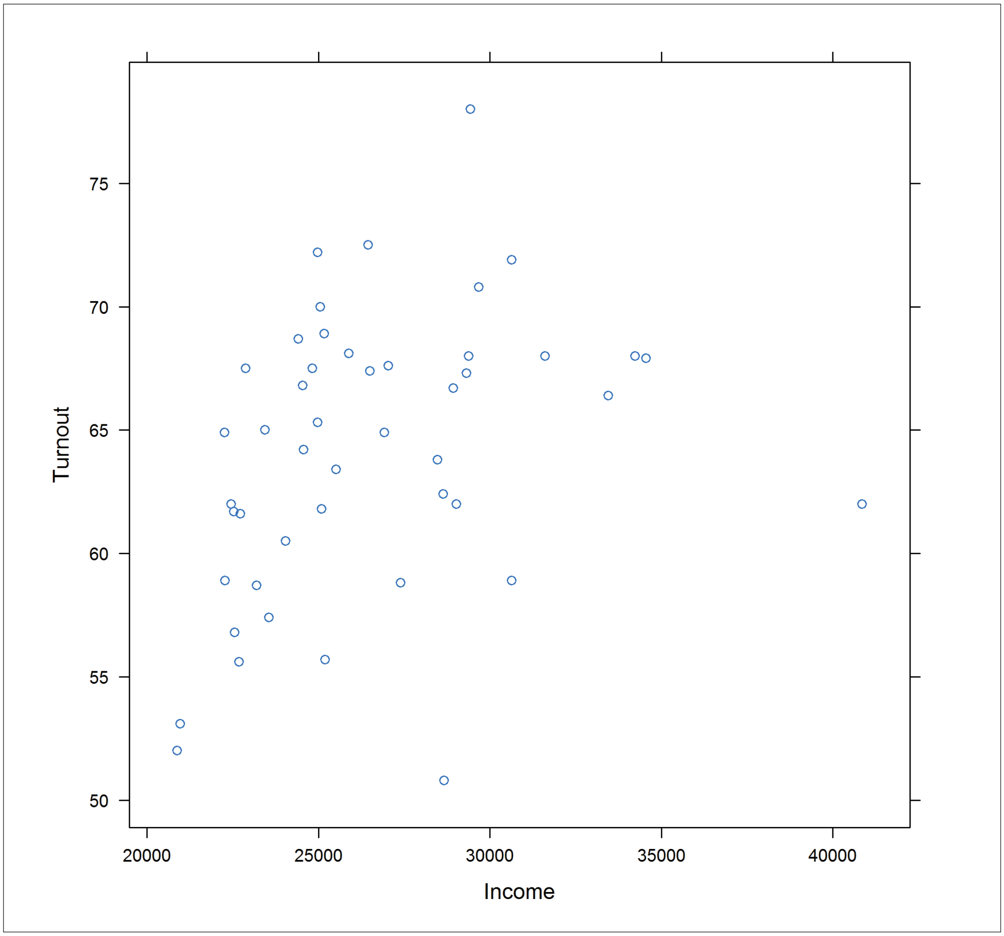


图 14-6：使用 lattice 系统绘制的简单散点图

很多用于改变绘图功能的选项与 base 系统基本相同。图 14-7 模仿图 14-2 改变了颜色和点的形状：

```
xyplot(Turnout ~ Income, obama_vs_mccain, col = "violet", pch = 20)
```

但是，轴的尺度需要以不同的方式指定。lattice 绘图接受一个 scales 参数，而且它必须是一个列表。这个列表里的内容必须是 name = value 对，例如，log = TRUE 为 x 和 y 轴设置了对数坐标。scales 列表还可以接受其他命名为 x 和 y 的（子）列表参数来指定其中所需的轴的设置。不要紧张，它没有看上去那么复杂。图 14-8 和 14-9 的例子分别显示不同坐标的轴：

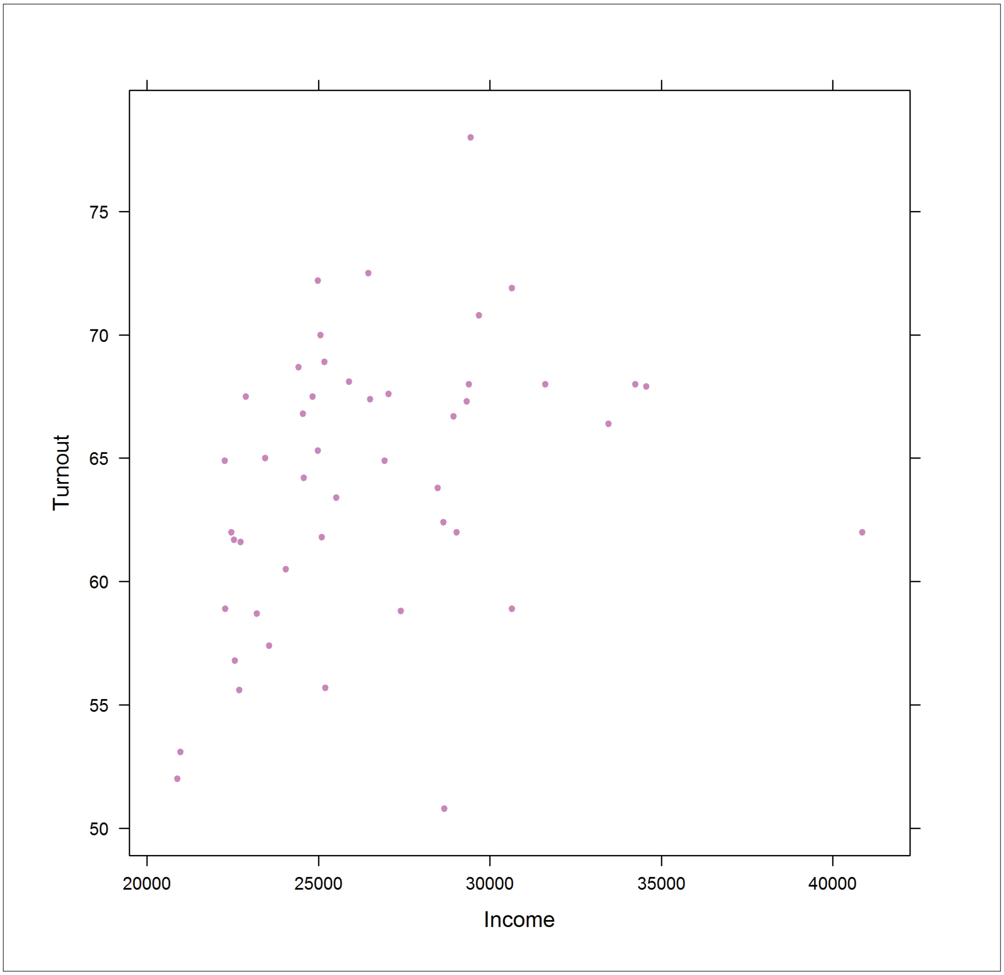


图 14-7：使用 lattice 系统设置颜色和点的形状

```
xyplot(
  Turnout ~ Income,
  obama_vs_mccain,
  scales = list(log = TRUE)              # x 和 y 轴都是对数坐标（图 14-8）
)

xyplot(
  Turnout ~ Income,
  obama_vs_mccain,
  scales = list(y = list(log = TRUE))    # y 轴对数坐标（图 14-9）
)
```

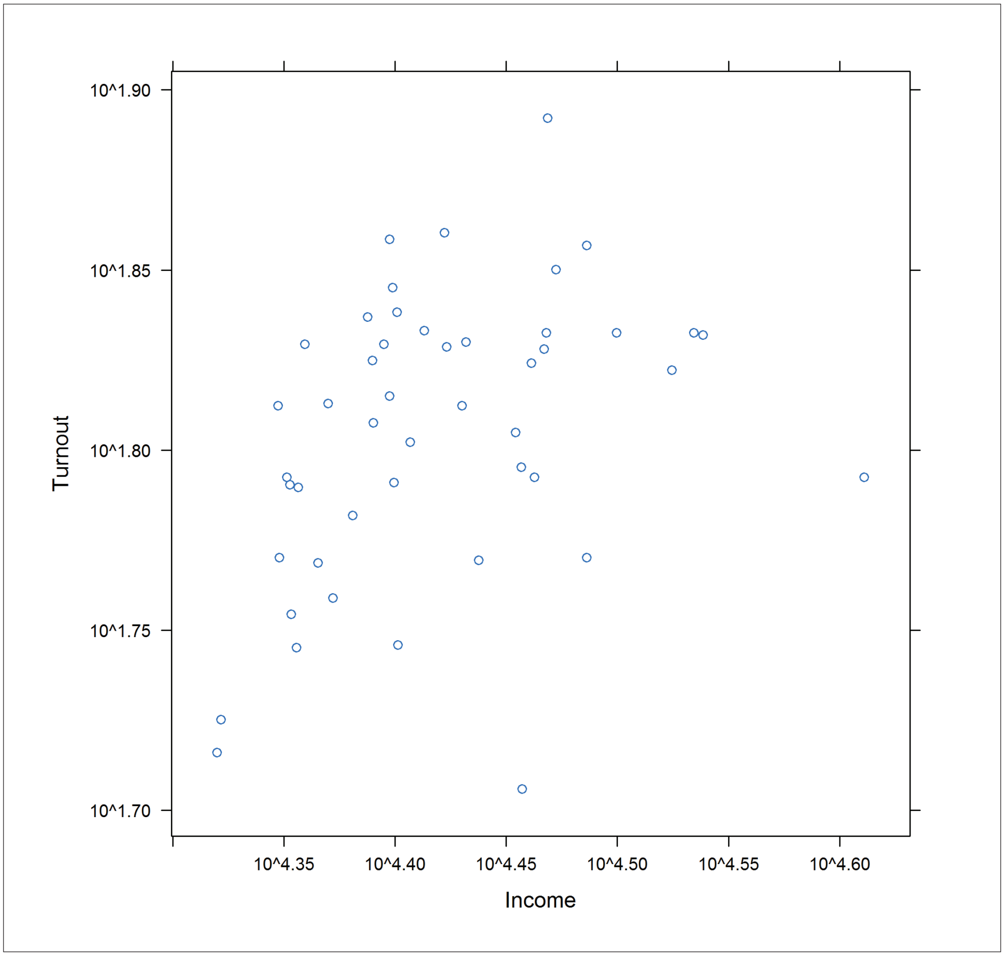


图 14-8：使用 lattice 系统的 x 和 y 的对数坐标

公式接口使按区域拆分数据变得很容易。我们只需要：追加一个 | 号（这是一个“管道”字符；与用于逻辑的“或”相同）和我们希望拆分的变量，这里即 Region。使用参数 relation = "same" 意味着每个面板都将使用相同的轴。当参数 alternating 为 TRUE（默认值）时，每个面板上轴的刻度将在绘图区的两侧交替出现，否则只出现在左侧和底部。输出如图 14-10 所示，请注意它对图 14-5 的改进：

```
xyplot(
  Turnout ~ Income | Region,
  obama_vs_mccain,
  scales = list(
```

```
    log         = TRUE,
    relation    = "same",
    alternating = FALSE
  ),
  layout = c(5, 2)
)
```

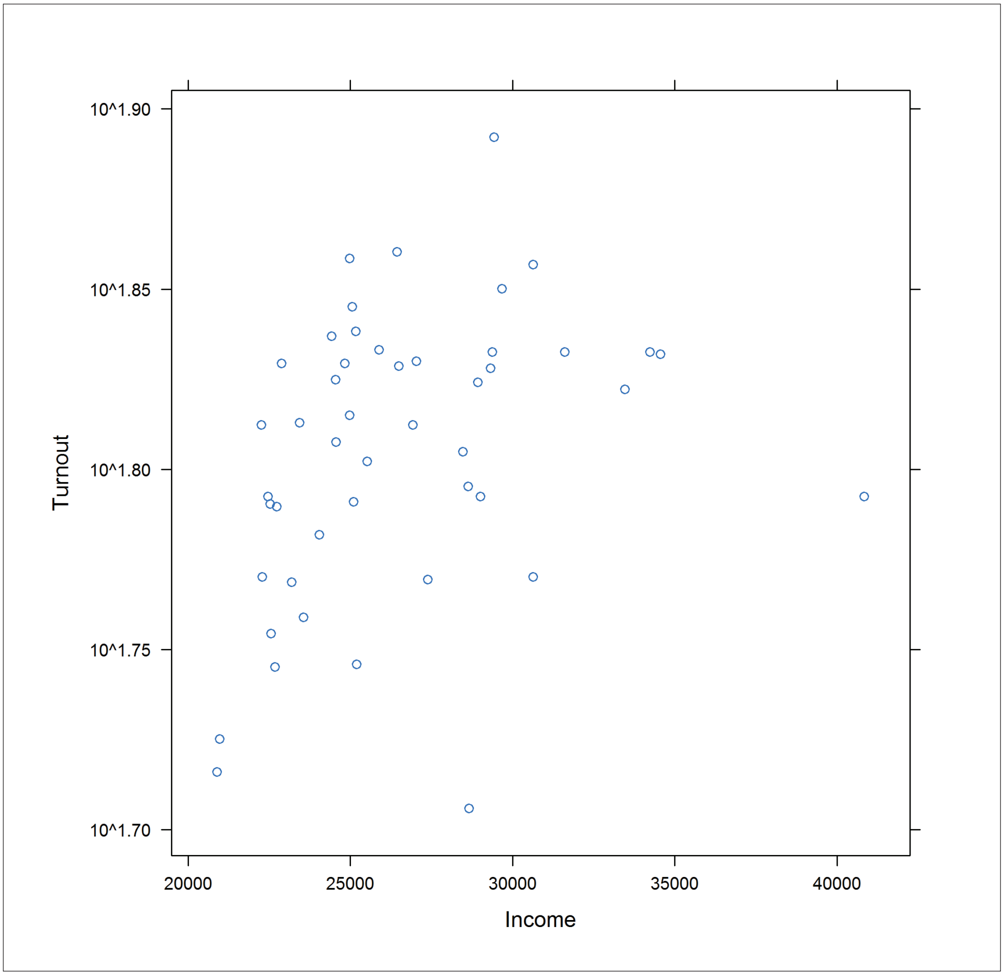


图 14-9：使用点阵图形登录 y 轴

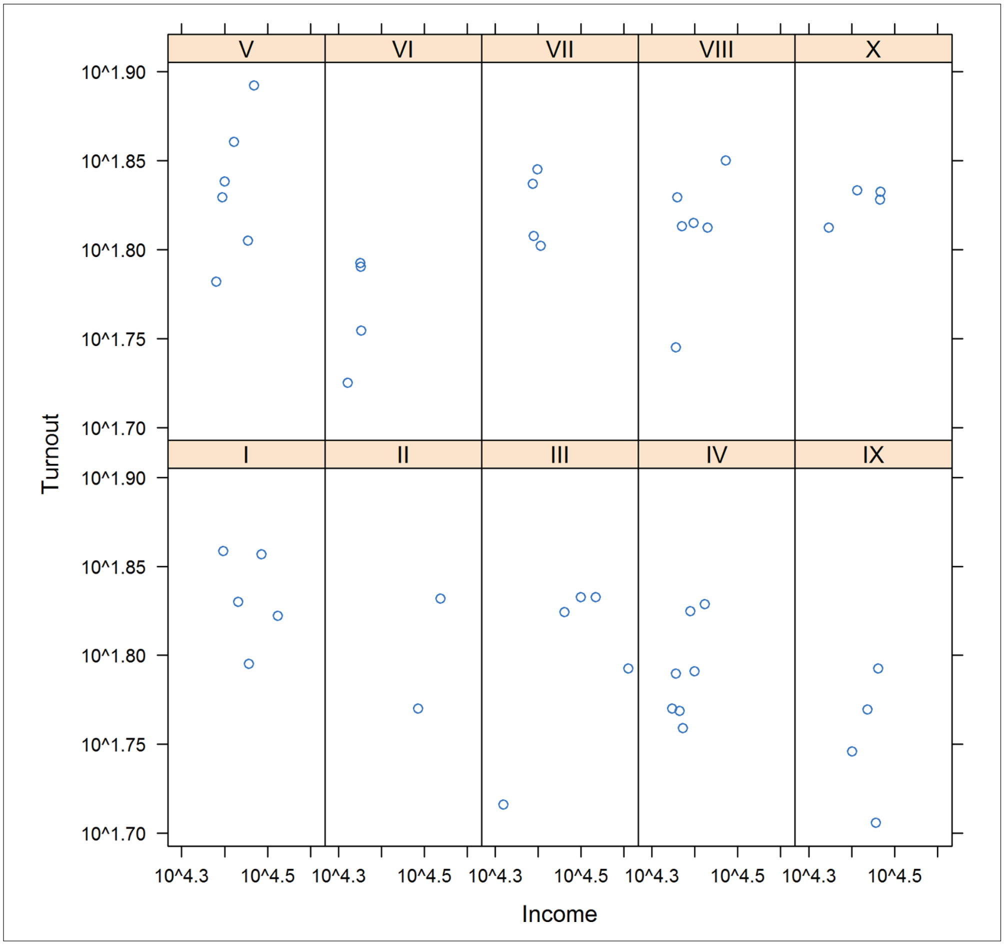


图 14-10：使用 lattice 系统在同一张图中绘制多条曲线

lattice 系统的另一个好处是它能把绘图存储在变量中，（ base 绘图与此相反，它只能把图绘制在窗口中），因而可在之后更改它们。图 14-12 中显示了图 14-11 的更改版本：

```
(lat1 <- xyplot(
  Turnout ~ Income | Region,
  obama_vs_mccain
))
# 图 14-11

(lat2 <- update(lat1, col = "violet", pch = 20))
# 图 14-12
```

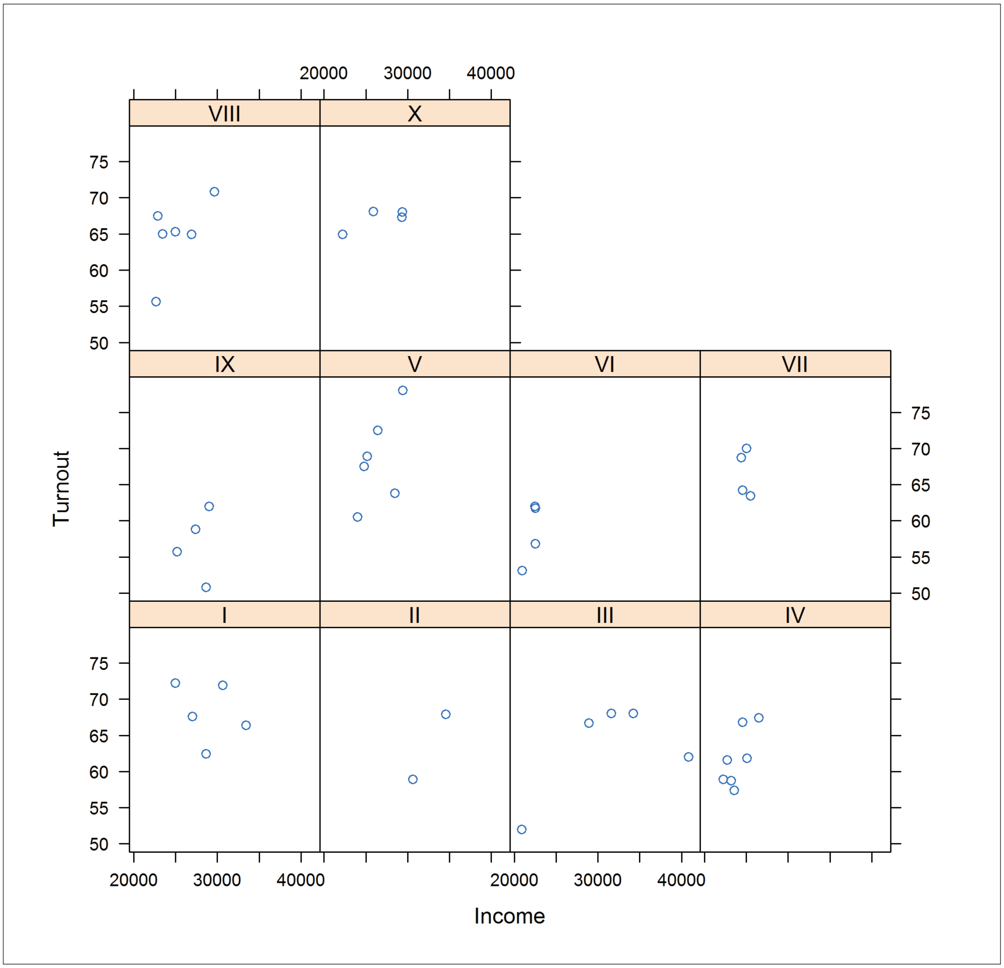


图 14-11：此绘图被存储为变量，它会在图 14-12 中被重用

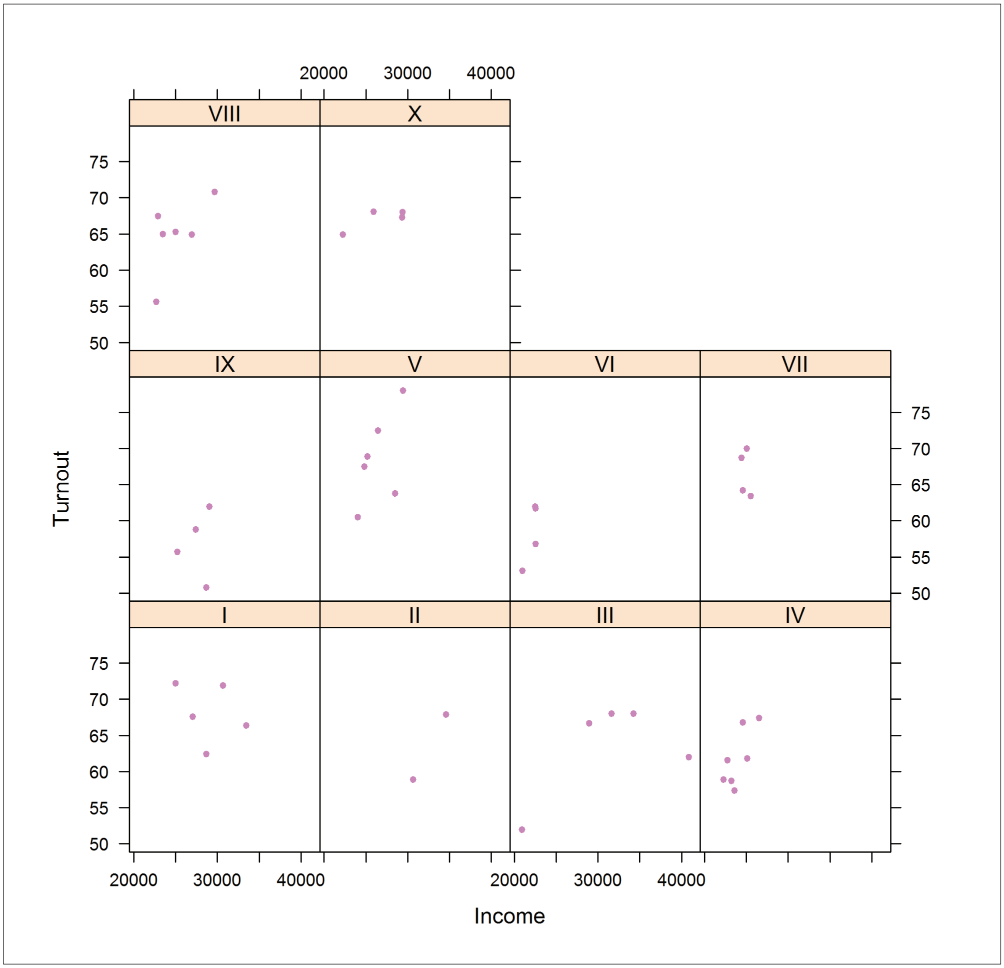


图 14-12：此图重用图 14-11 中的 `lattice` 系统变量

14.4.3 第三种方法：ggplot2图形系统

`ggplot2`（“2”是因为尝试了多次才得到理想的版本）从 `lattice` 系统中吸收了很多好点子，并基于它所建立。因此，将绘图拆分成多个面板非常容易，且能够按顺序创建图形。除此之外，`ggplot2` 有它自己的一些特殊技巧。最重要的是，它本质上是“语法的”，这意味着它由众多小的组件构成，因此它更容易创建全新的绘图类型，如果你特想这么做的话。

它的语法与其他系统的代码非常不同，要做好思想准备来接受新事物。每个绘图由 `ggplot` 函数创建，它的第一个参数是一个数据框，第二个是 *aesthetic*。其实就是把 `x` 和 `y` 列变量传递给 `aes` 函数。然后，我们再添加一个 *geom* 让图形系统显示一些点。图 14-13 显示了结果：

```
library(ggplot2)
ggplot(obama_vs_mccain, aes(Income, Turnout)) +
  geom_point()
```

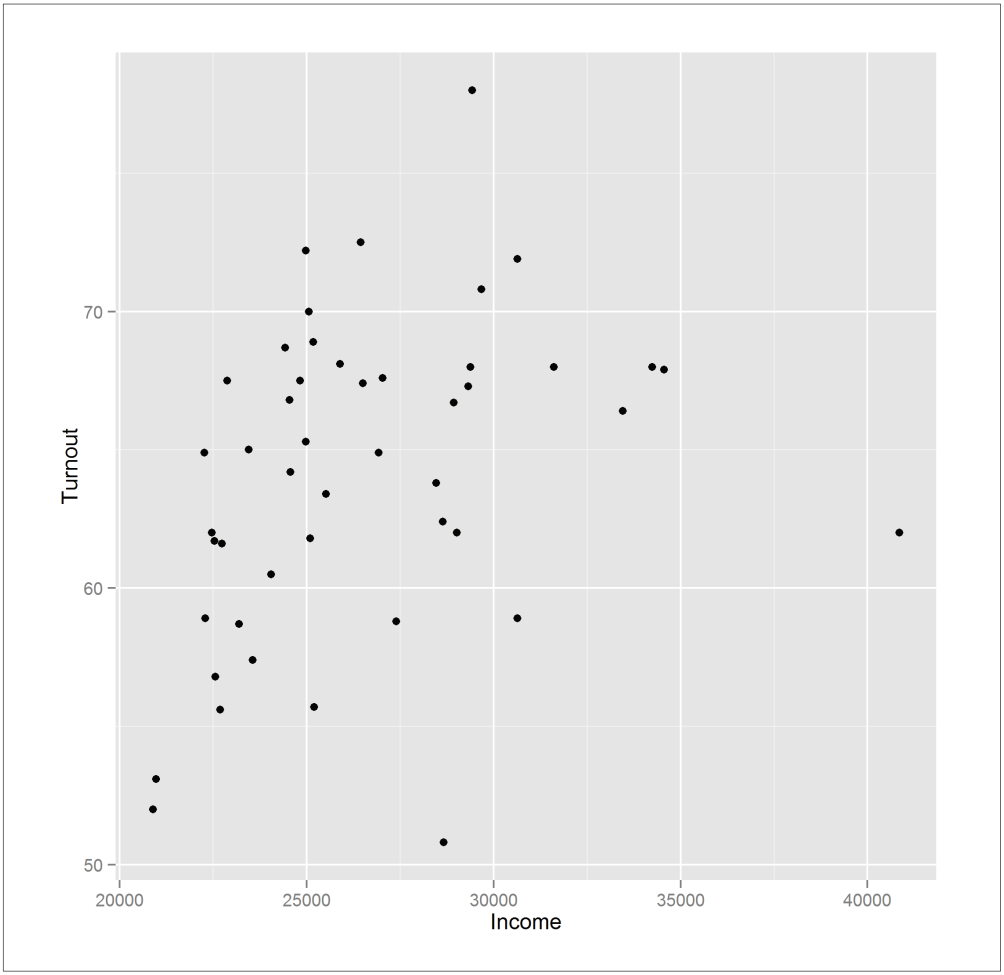


图 14-13：使用 ggplot2 制图法绘制简单的散点图

ggplot2 不仅能识别来自 base 图形系统的命令来改变点的颜色和形状，而且也有它自己的一套更加可读的名称。在图 14-14 中，shape 取代了 pch，且颜色可使用 color 或 colour 来指定：

```
ggplot(obama_vs_mccain, aes(Income, Turnout)) +
  geom_point(color = "violet", shape = 20)
```

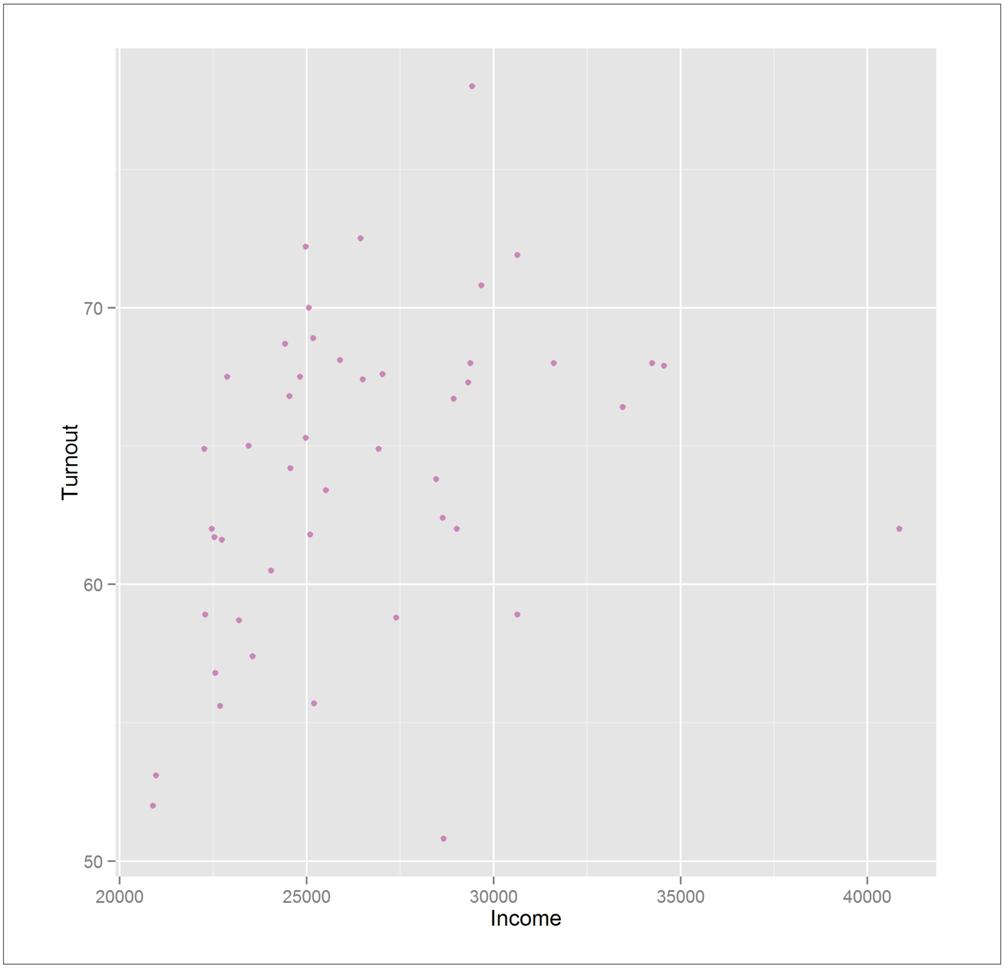


图 14-14：使用 ggplot2 绘图法设置点的颜色和形状

为了设置一个对数坐标，我们需要为每个轴添加一个标度，如图 14-15 所示。break 参数指定了轴刻度值的位置。它是可选的，但这里用来复制 base 和 +lattice 系统范例中的行为：

```
ggplot(obama_vs_mccain, aes(Income, Turnout)) +
  geom_point() +
  scale_x_log10(breaks = seq(2e4, 4e4, 1e4)) +
  scale_y_log10(breaks = seq(50, 75, 5))
```

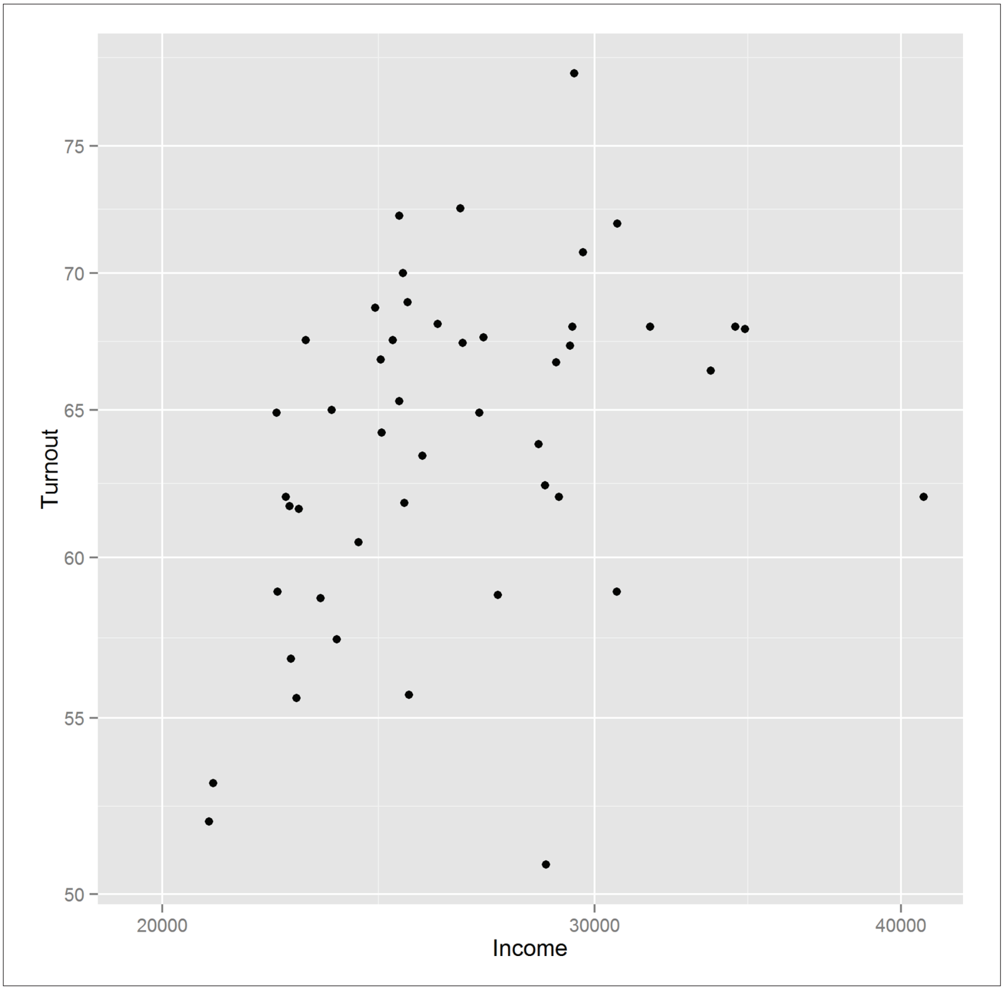


图 14-15：使用 ggplot2 绘制对数坐标

为了把绘图分割成不同的面板，我们添加一个切面（facet）。与 lattice 中的绘图类似，切面带有一个公式参数。图 14-16 显示了 facet_wrap 函数的行为。为方便阅读，x 轴的刻度已被旋转了 30 度，并用 theme 函数使其右对齐：

```
ggplot(obama_vs_mccain, aes(Income, Turnout)) +
  geom_point() +
  scale_x_log10(breaks = seq(2e4, 4e4, 1e4)) +
  scale_y_log10(breaks = seq(50, 75, 5)) +
  facet_wrap(~ Region, ncol = 4)
```

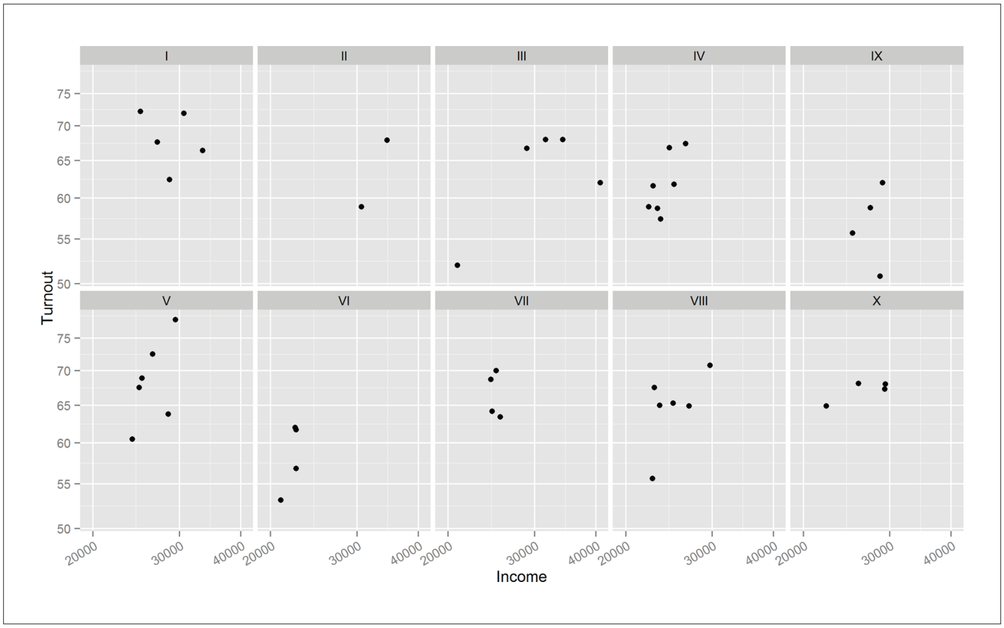


图 14-16：使用 ggplot2 把多个图形画在同一张图中

为了将多个变量分开来，我们把公式指定为类似 `~ var1 + var2 + var3` 的形式。对于只有两个变量的特殊情况，`facet_grid` 提供了另一种方法来分开它们，一个置于行中，而另一个置于列中。

与 `lattice` 类似，`ggplots` 可以把图形存储到变量中，然后继续往它上面添加东西。下例重新绘制了图 14-13 并把它存储为一个变量。和往常一样，把表达式置于括号中能使它自动打印出来：

```
(gg1 <- ggplot(obama_vs_mccain, aes(Income, Turnout)) +
  geom_point()
)
```

图 14-17 显示了其输出。我们可以使用以下代码更新它，并在图 14-18 显示其结果：

```
(gg2 <- gg1 +
  facet_wrap(~ Region, ncol = 5) +
  theme(axis.text.x = element_text(angle = 30, hjust = 1))
)
```

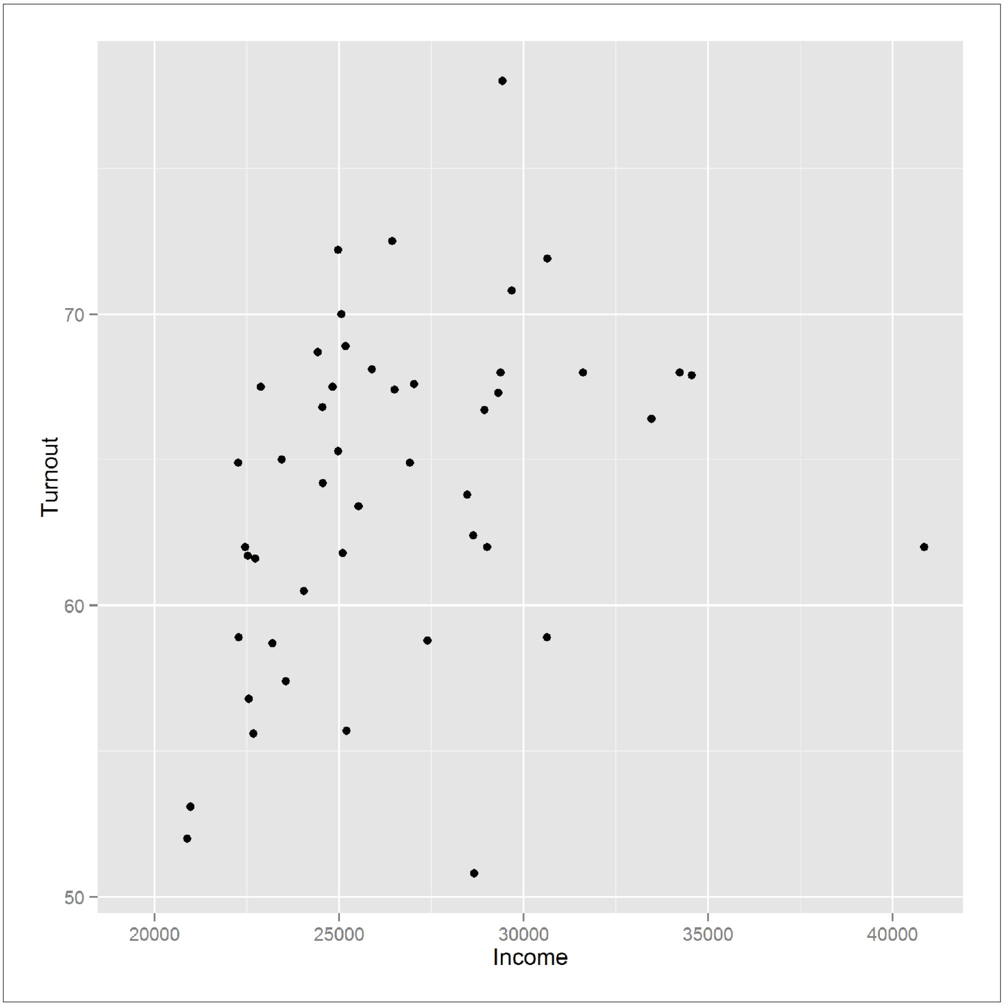


图 14-17：此图被存储为变量并将在图 14-18 中重用

14.5 线图

如果想研究连续变量如何随时间变化，线图往往比散点图更加容易理解，因为它能显示邻近数值之间的联系。下例将在 `crab_tag` 数据集中研究螃蟹一年中的情况，看看它们曾游到北海多深的位置。

在 `base` 图形系统中，线图与散点图的创建方式一样，不同的是线图采用参数 `type = "l"`。为了避免在维度上的混淆，我们把深度画成负数，而不是使用给定数据集中的绝对值。

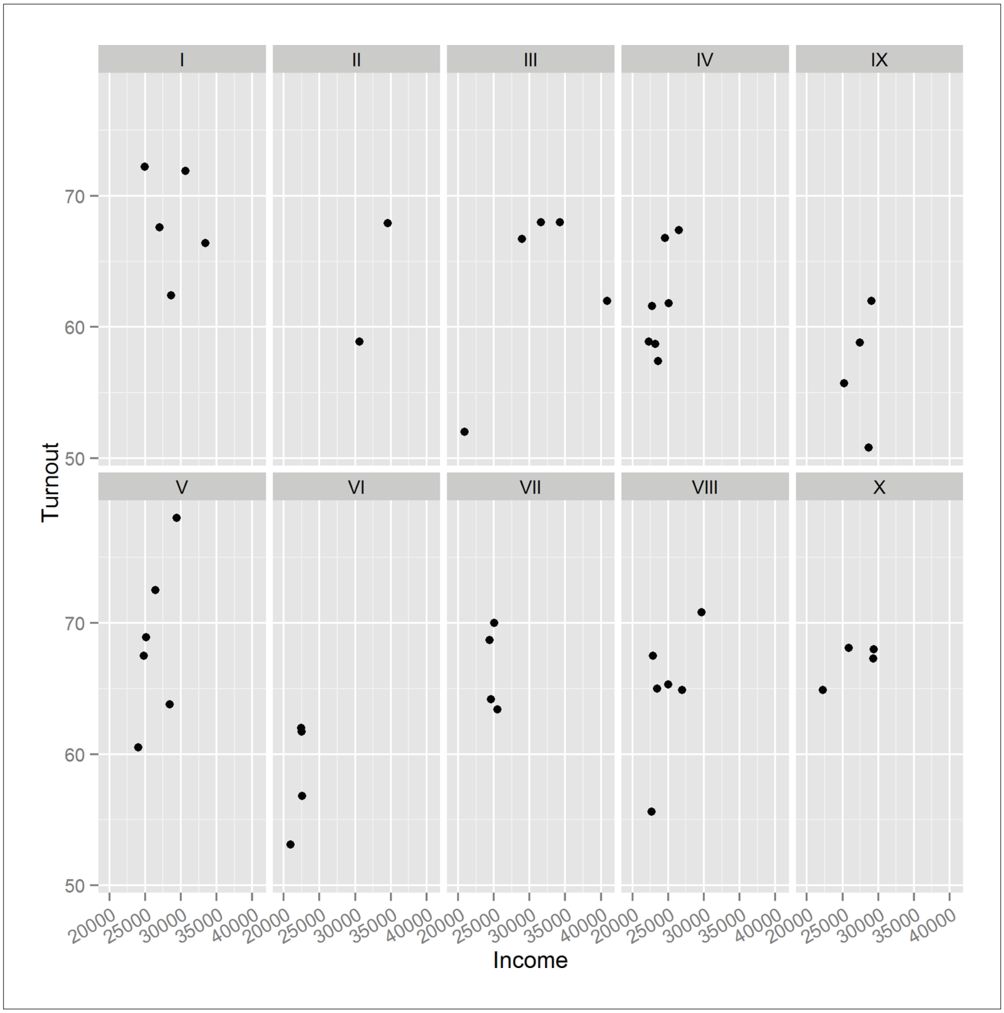


图 14-18：此图重用了从图 14-17 中得到的 `ggplot2` 变量

绘图区范围将被默认设为数据的区间范围（但会加多一点点，更多细节请参见 `?par` 帮助页面的 `xaxs` 一节）。为了获得更好的透视感，我们会通过传统 `ylim` 参数来手动设置 y 轴的大小，使它的范围为螃蟹在海中走到的最深点到海平面之间。图 14-19 显示了这个线图：

```
with(
  crab_tag$daylog,
  plot(Date, -Max.Depth, type = "l", ylim = c(-max(Max.Depth), 0))
)
```

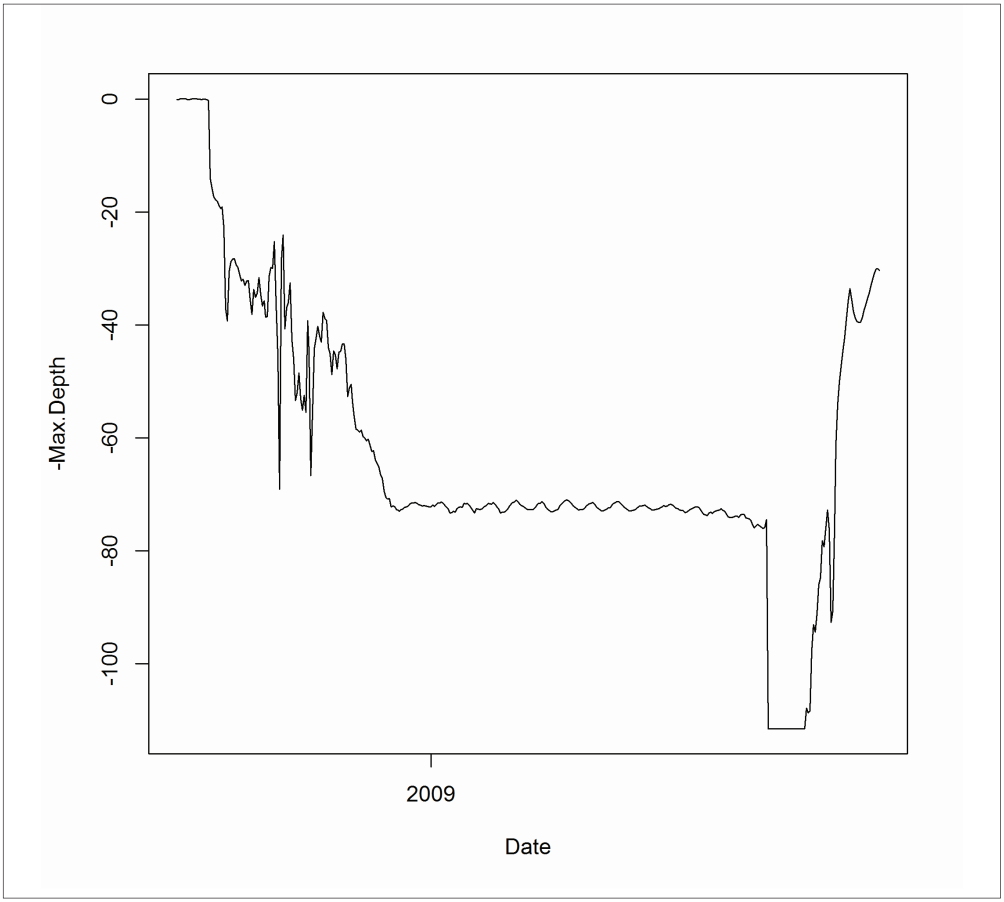


图 14-19：使用 base 制图法绘制的线图

现在，故事才只讲了一半。Max.Depth 参数是螃蟹在某一天所达到的大海最深处。我们还需要为线图添加一个 Min.Depth，目的是为了方便查看螃蟹每天到达的最浅处。附加线可使用 lines 函数在现有的绘图中重叠绘出。对于散点图，类似的函数是 points。图 14-20 显示了另一条线：

```
with(
  crab_tag$daylog,
  lines(Date, -Min.Depth, col = "blue")
)
```

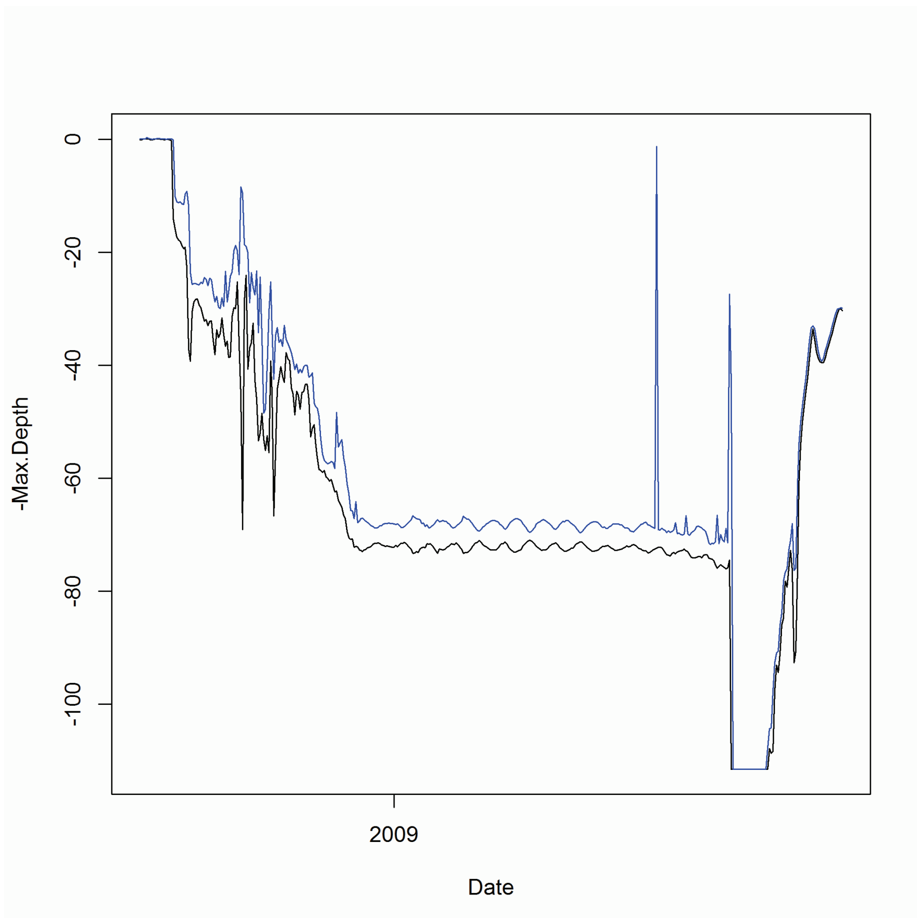


图 14-20：使用 base 制图法添加另一条线

lattice 与 base 系统遵循类似的模式。与散点图一样，lattice 也使用 xyplot 来画线图，但同样也要使用 type ="l" 的参数。使用公式接口能轻而易举地指定多行。注意，公式中的 + 号常用于创建类似于图 14-21 的图：

```
xyplot(-Min.Depth + -Max.Depth ~ Date, crab_tag$daylog, type = "l")
```

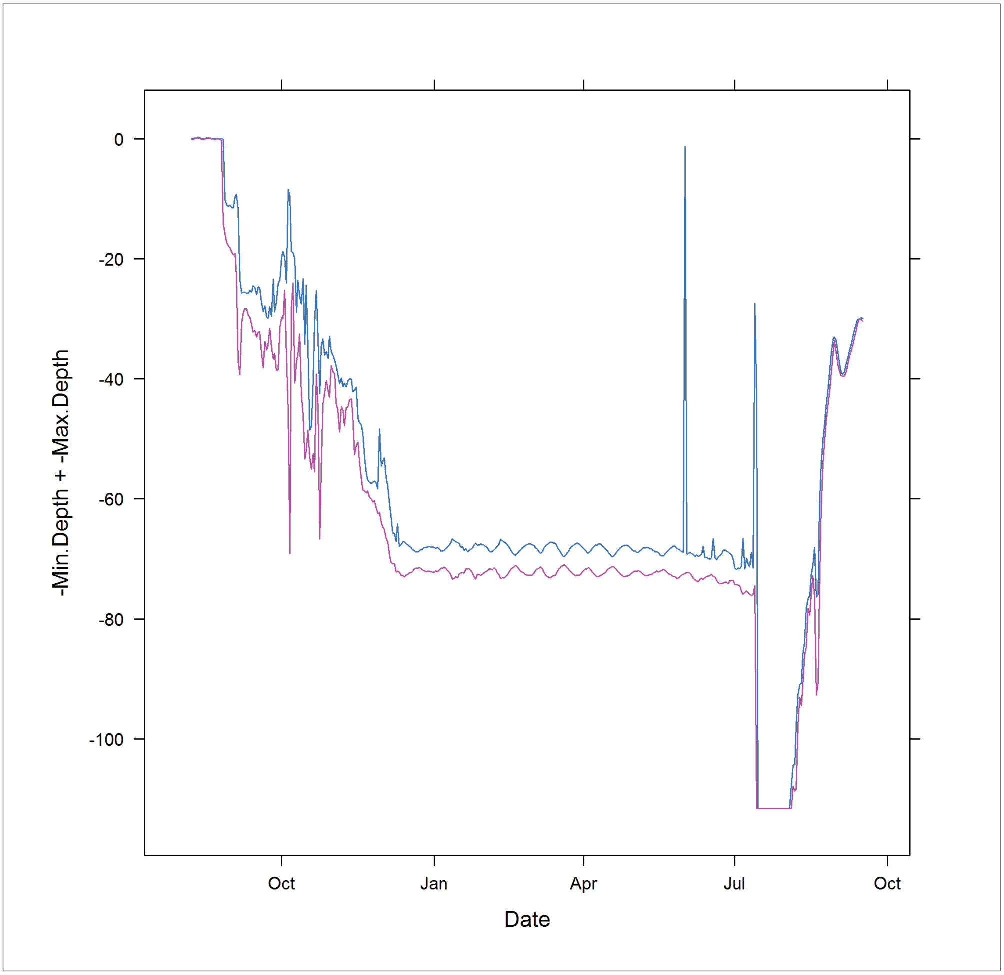


图 14-21：使用 lattice 图形系统绘制的线图

在 ggplot2 中从散点图切换到线图非常简单，只要把 geom_plot 替换为 geom_line 即可（图 14-22 显示了结果）：

```
ggplot(crab_tag$daylog, aes(Date, -Min.Depth)) +
  geom_line()
```

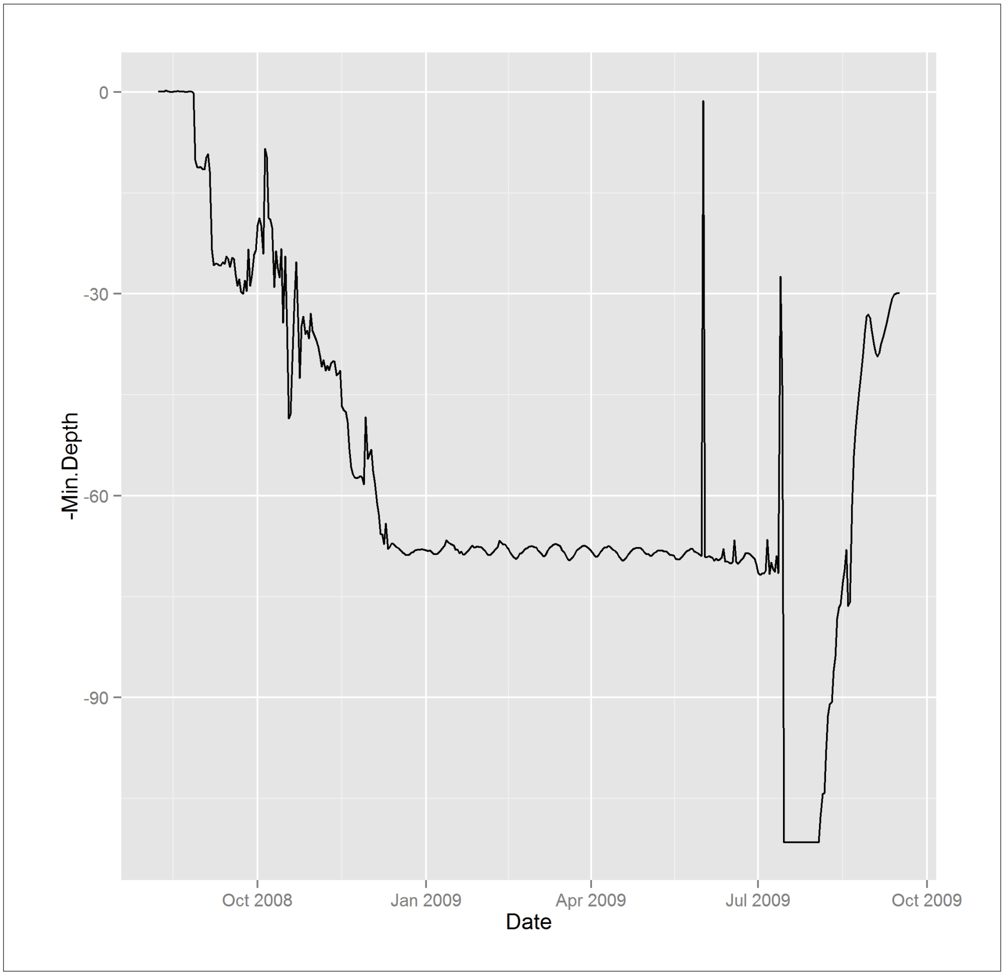


图 14-22：使用 ggplot2 系统绘制的线图

不过，在绘制多条线时会有一点复杂。当你在 `ggplot` 指定 aesthetics 时，你会为所有的 `geom` 都指定了这个 aesthetics。也就是说，aesthetics 对绘图来说其影响是“全局的”。在下例中，我们希望在一条线上指定最大深度，而在另一条线上指定最小深度，如图 14-23 所示。一种解决方法是在每个 `geom_line` 函数中指定一个 y-aesthetic：

```
ggplot(crab_tag$daylog, aes(Date)) +
  geom_line(aes(y = -Max.Depth)) +
  geom_line(aes(y = -Min.Depth))
```

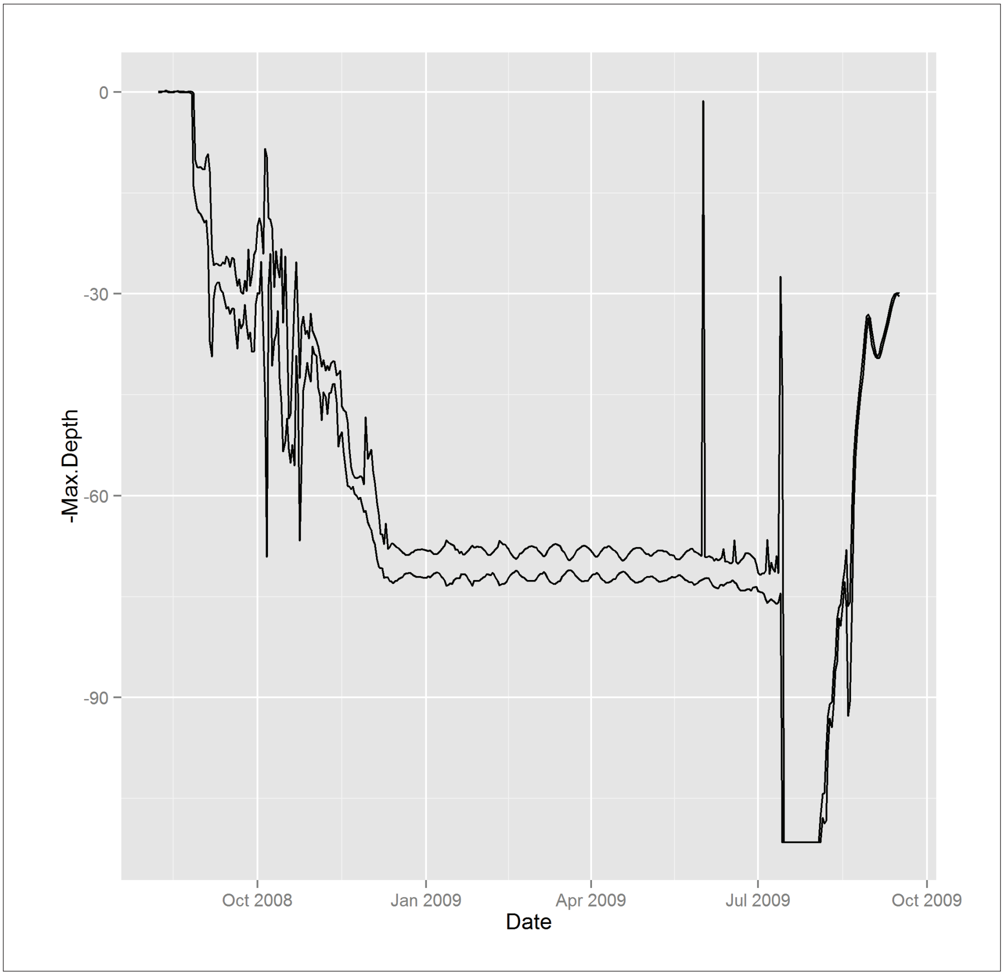


图 14-23：使用 ggplot2 绘制的两条具有独立 geoms 的线

然而，这有点笨拙，因为需要调用 geom_line 两次，且实际上这也并非惯常的做法。在 ggplot2 中，更“恰当的”方式是将数据熔化（melt）成长表，然后把线分组（group），如图 14-24 所示：

```
library(reshape2)
crab_long <- melt(
  crab_tag$daylog,
  id.vars      = "Date",
  measure.vars = c("Min.Depth", "Max.Depth")
)
ggplot(crab_long, aes(Date, -value, group = variable)) +
  geom_line()
```

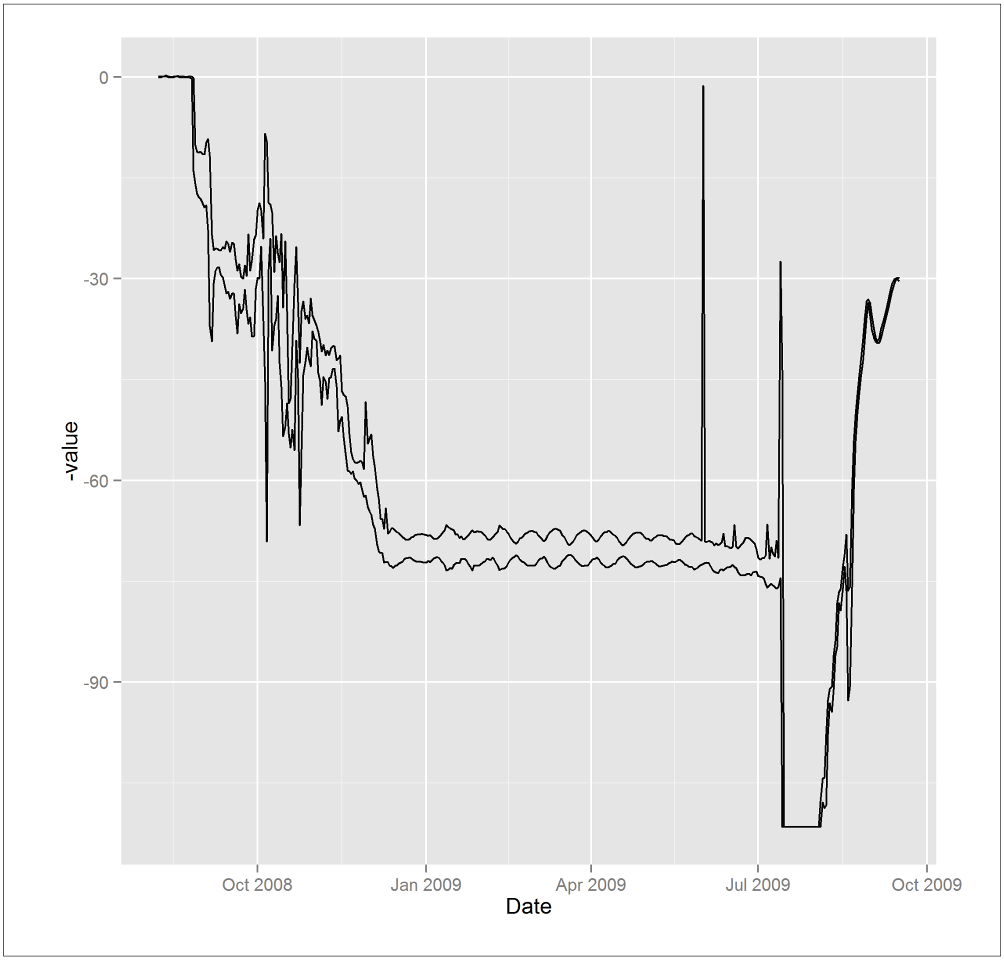


图 14-24：使用 ggplot2 和分组绘制两条线

对于只有两条线的场景，有一个更好的解决方案，它不需要任何的数据操作。geom_ribbon 将绘制两条直线以及它们中间的内容。为了使图形更美观，可把 color 和 fill 参数传递给 geom，以此指定线的颜色和填充形式。图 14-25 显示了结果：

```
ggplot(crab_tag$daylog, aes(Date, ymin = -Min.Depth, ymax = -Max.Depth)) +
  geom_ribbon(color = "black", fill = "white")
```

无论你使用哪种系统来绘图，螃蟹的行为都是非常明确的。9 月，它生活在浅水域准备交配，然后花了几个月的时间迁移到深水区。在整个冬季、春季和夏季它都愉快地行走在北海海底（除了在 6 月初，它在水平面有一次奇怪而短暂的旅行——不知道是数据错误，还是它从渔船中捡回一条命）。之后，在 7 月中旬它显然从“悬崖”掉了下来，在爬回浅水区开始下一轮的交配之前被捕获了。

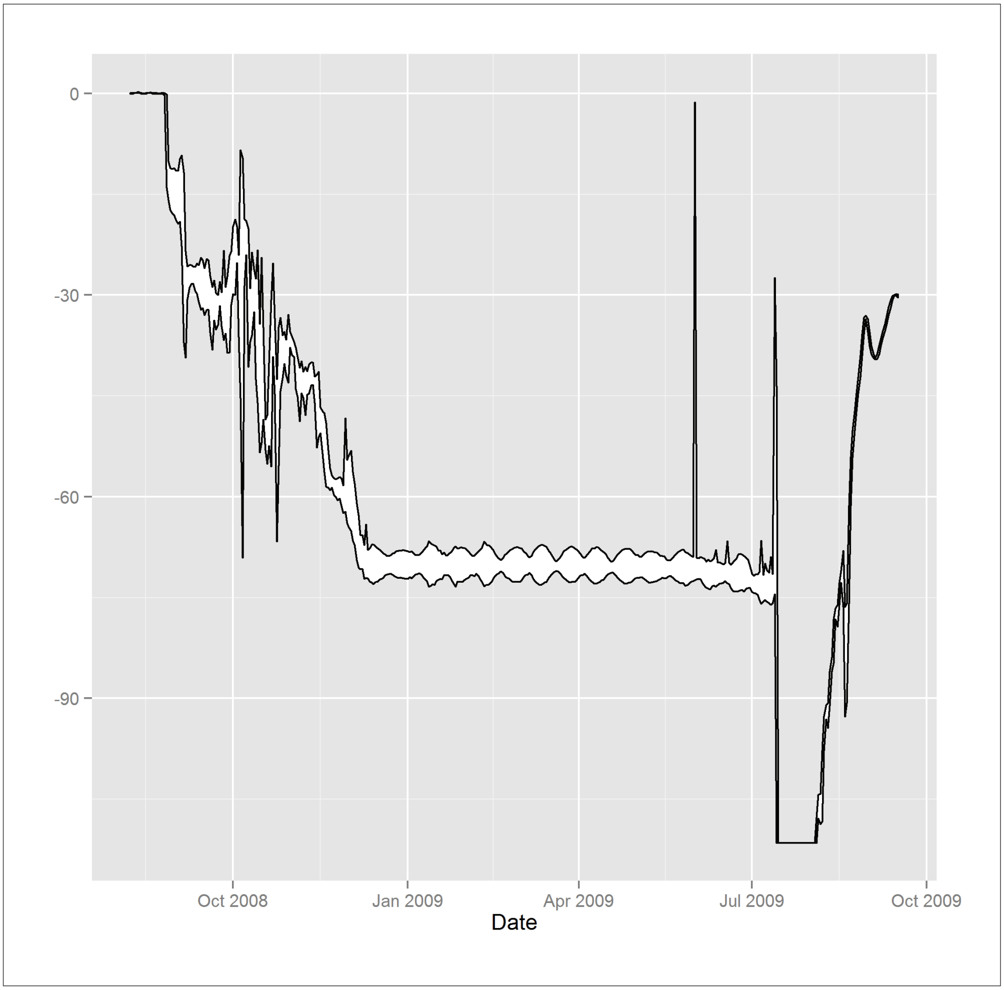


图 14-25：使用 `ggplot2` 图形的彩带图（ribbon plot）

14.6 直方图

如果你要研究一个连续变量的分布，直方图是最佳选择[6]。

在下例中，我们将返回到 `obama_vs_mccain` 数据集，这次只研究奥巴马的得票比例分布。在 `base` 中可以使用 `hist` 函数绘制直方图，如图 14-26 所示。与 `plot` 类似，它没有 `data` 参数，须把它置于 `with` 内：

注 6：Dataviz 的狂热粉丝常认为核密度图（kernel density plot）对于这种分布来说会有“更好”的表现形式。但缺点是每次向没学过统计的人展示时，得先花上 15 分钟解释什么是核密度图。

```
with(obama_vs_mccain, hist(Obama))
```

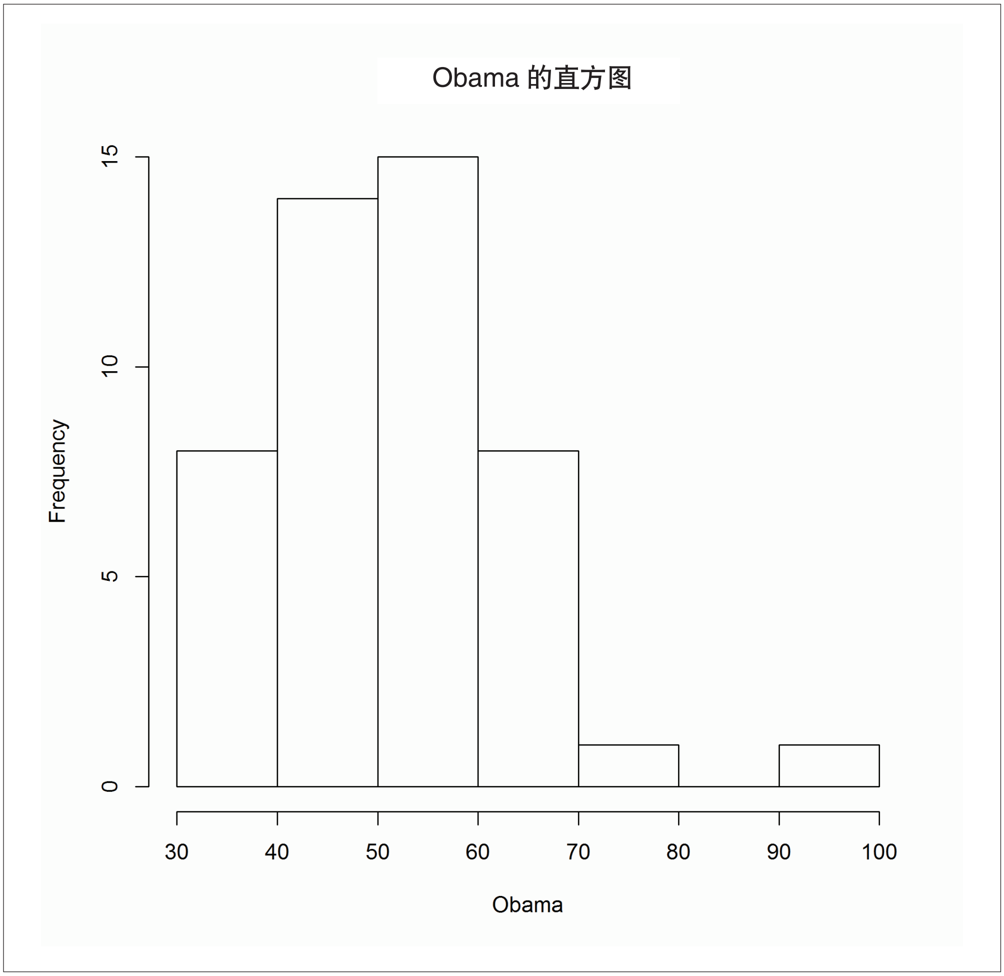


图 14-26：使用 base 系统绘制的直方图

直方图的区间间距数值默认由斯特奇斯（Sturges）算法进行计算。你可以实验着使用各种不同的子区间（bin）宽度，这样对分布的理解将更完整。这非常灵活，可以通过多种方式来实现：给 `hist` 传递一个数字来指定区间的数目，或一个区间边缘的向量，或用于计算区间数目的算法名称（除了默认的 `"sturges"` 之外，目前还支持 `"scott"` 和 `"fd"`），或一个能计算前两个选项之一的函数。在下例中，其结果如图 14-27 至 14-31 所示，`main` 参数创建了图中的主标题。这也同样适用于 `plot` 函数：

```
with(obama_vs_mccain,
  hist(Obama, 4, main = "An exact number of bins")
)
# 图 14-27
```

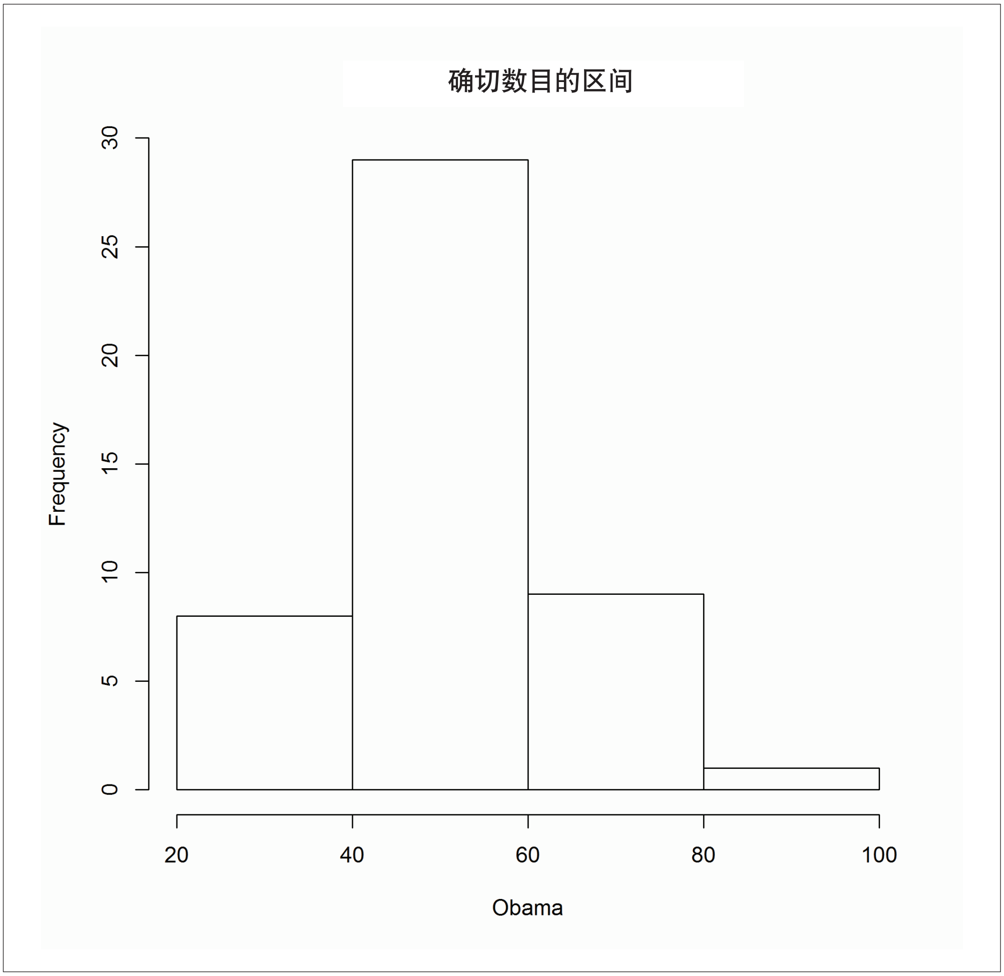


图 14-27：在 base 系统中为直方图指定一个确切数目的区间间距

```
with(obama_vs_mccain,
  hist(Obama, seq.int(0, 100, 5), main = "A vector of bin edges")
)
# 图 14-28
```

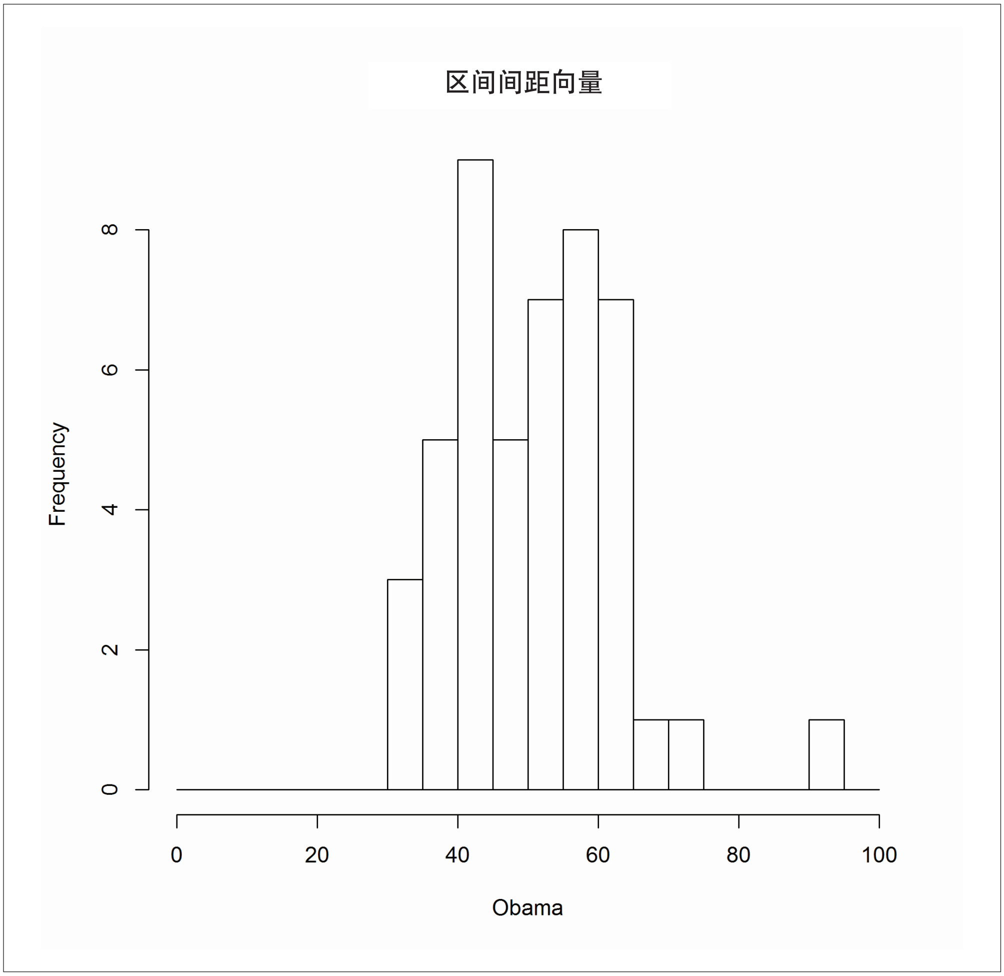


图 14-28：在 base 系统中为直方图指定一个确切数目的区间间距

```
with(obama_vs_mccain,
  hist(Obama, "FD", main = "The name of a method")
)
# 图 14-29
```

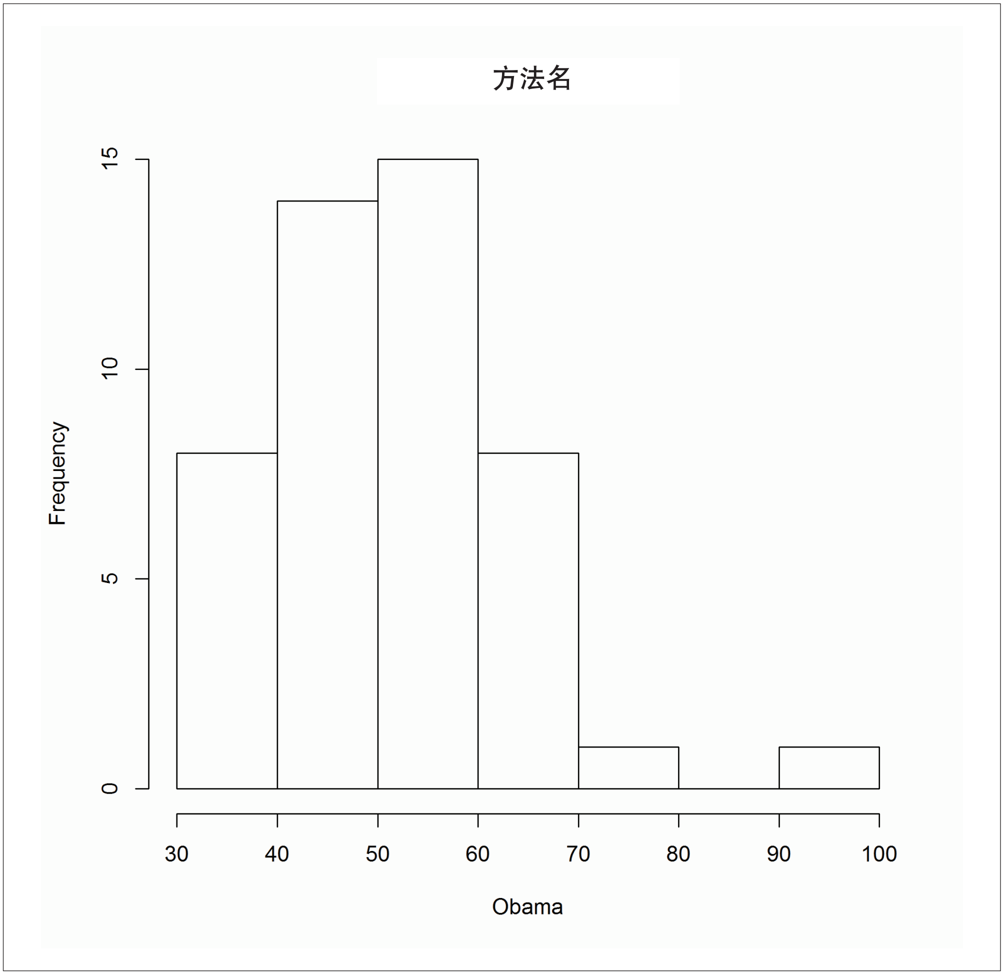


图 14-29：在 base 系统中使用算法名称为直方图指定区间间距

```
with(obama_vs_mccain,
  hist(Obama, nclass.scott, main = "A function for the number of bins")
)
# 图 14-30
```

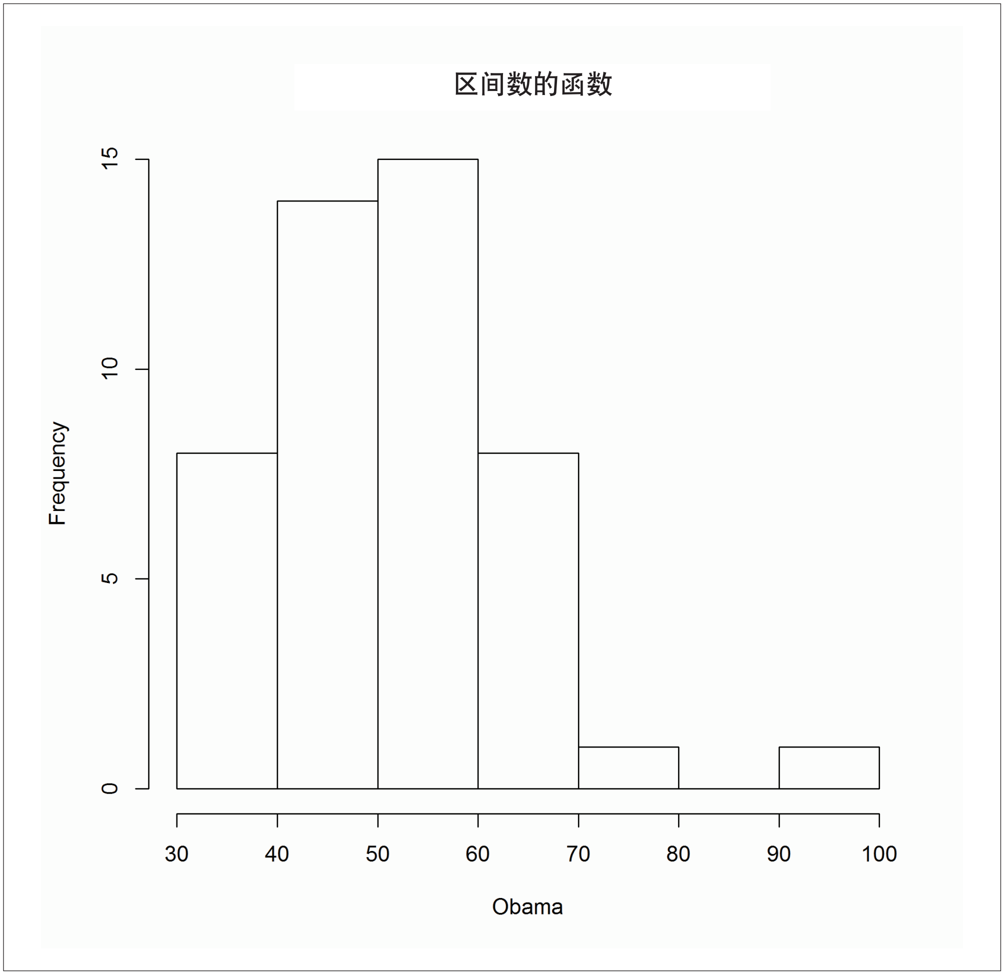


图 14-30：在 base 系统中使用返回区间数的函数为直方图指定区间间距

```
binner <- function(x)
{
  seq(min(x, na.rm = TRUE), max(x, na.rm = TRUE), length.out = 50)
}
with(obama_vs_mccain,
  hist(Obama, binner, main = "A function for the bin edges")
)
# 图 14-31
```

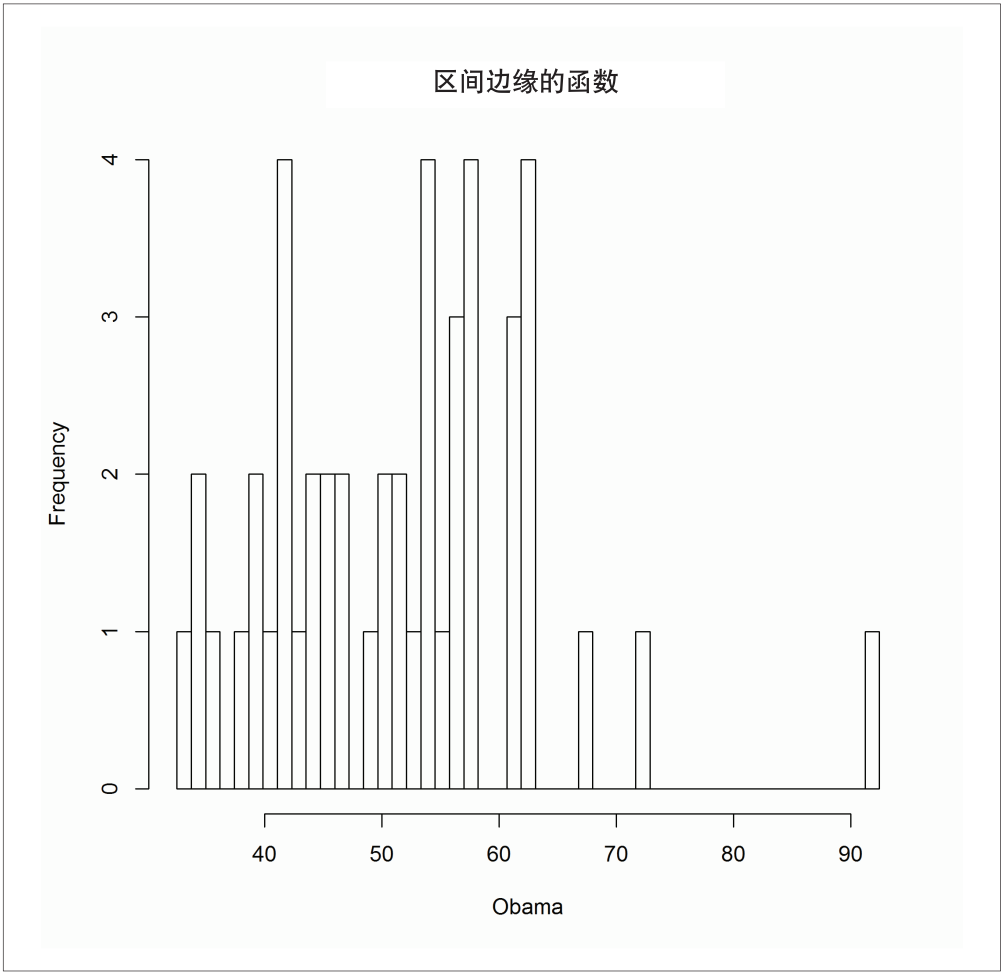


图 14-31：在 base 系统中使用返回区间边缘的函数为直方图指定区间间距

freq 参数控制直方图是否显示计数或显示每个区间的概率密度。当且仅当区间是均匀分布时，它默认为 TRUE。图 14-32 显示其输出：

```
with(obama_vs_mccain, hist(Obama, freq = FALSE))
```

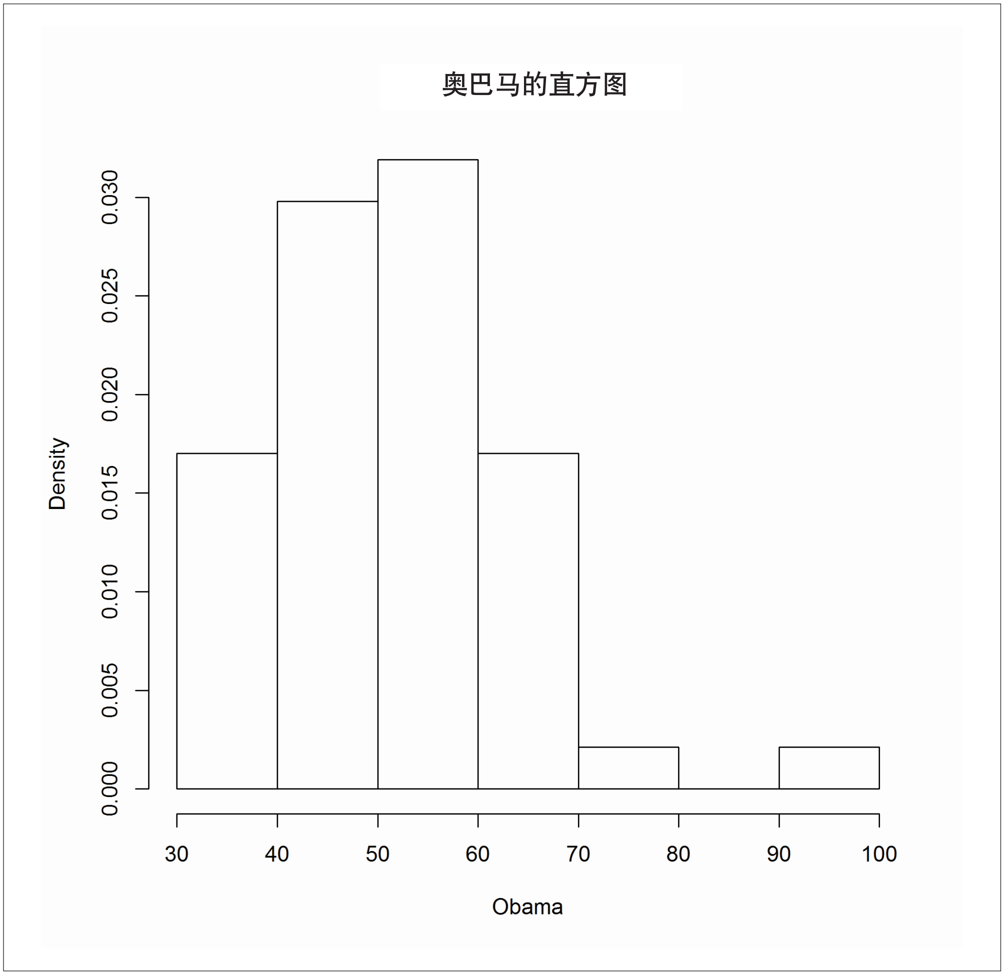


图 14-32：使用 base 系统绘制的概率密度直方图

lattice 中的直方图与 base 的类似，不过它还使用了一个 data 参数，这个参数使它能更易于分割成多个面板，且能把绘图保存为变量。breaks 参数与其在 hist 中的使用方式相同。图 14-33 和 14-34 显示了 lattice 直方图及其区间间距的规格：

```
histogram(~ Obama, obama_vs_mccain)
# 图 14-33

histogram(~ Obama, obama_vs_mccain, breaks = 10)
# 图 14-34
```

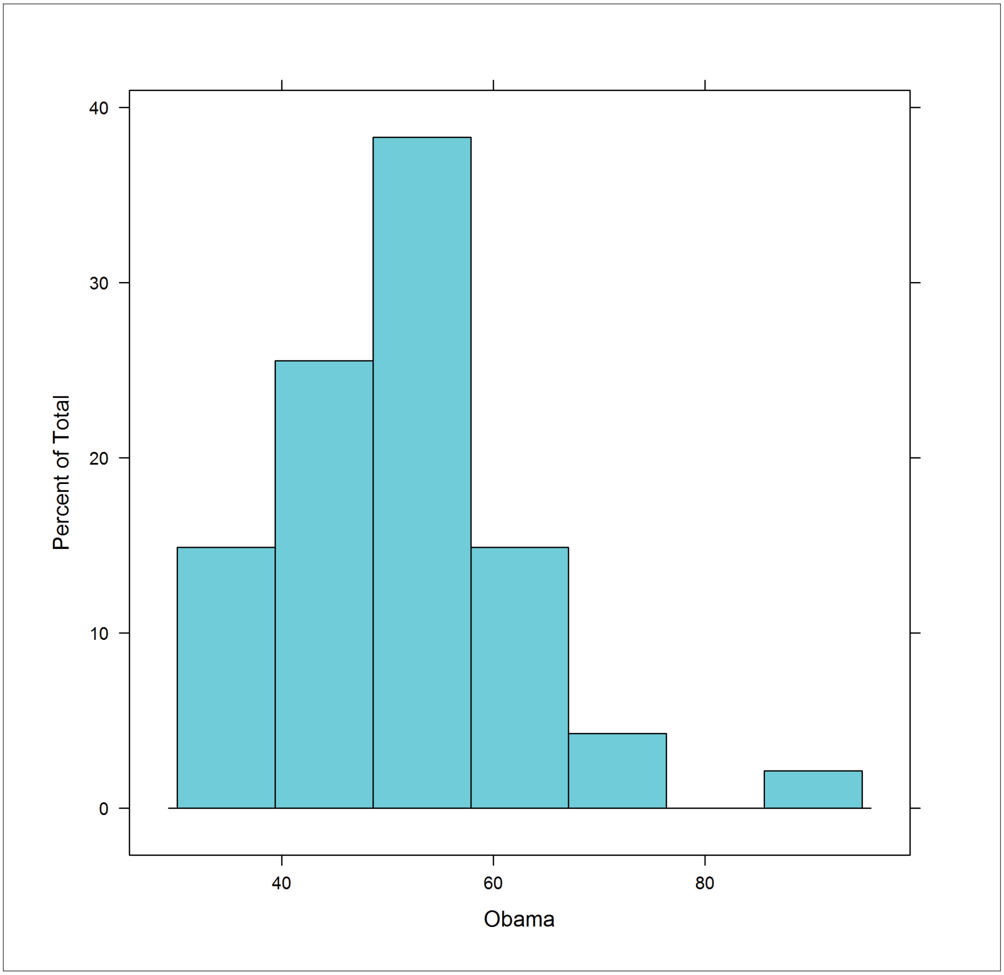


图 14-33：使用 `lattice` 系统绘制的直方图

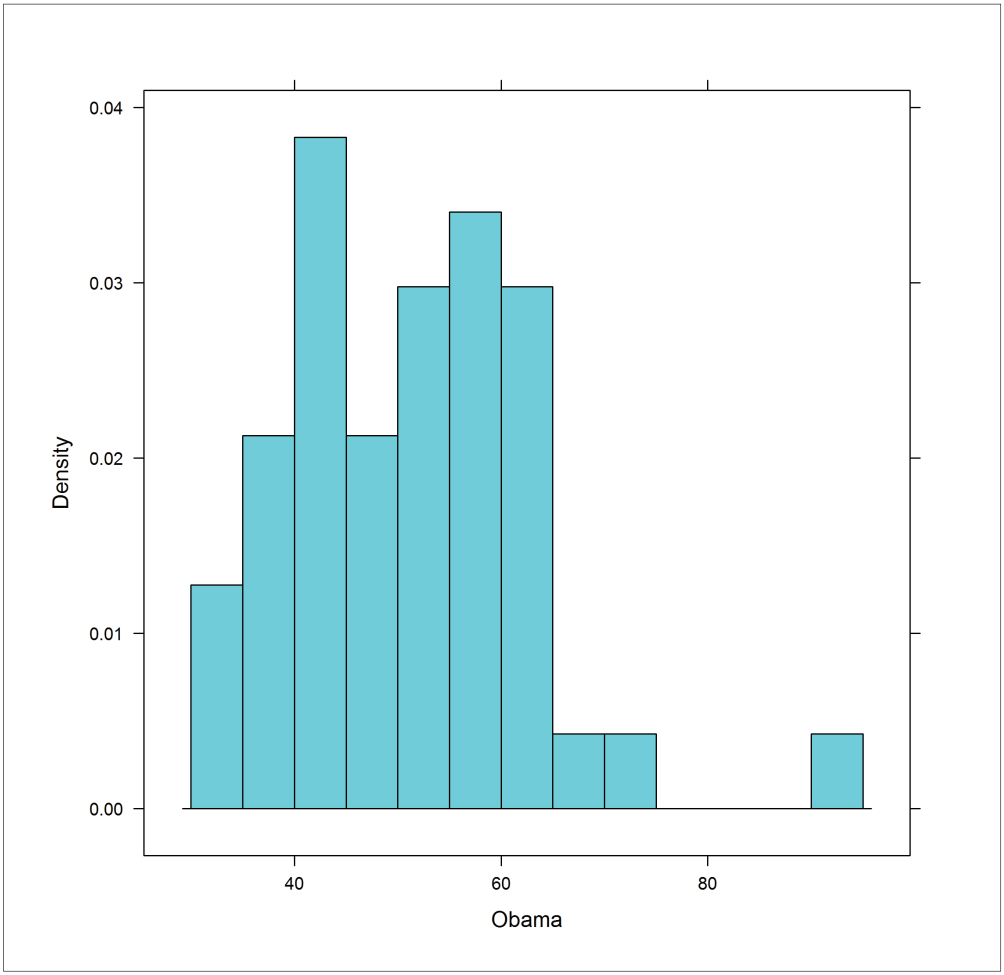


图 14-34：在 lattice 系统中指定区间间距

lattice 中的直方图能通过指定接收字符串 "count"、"density" 或 "percent" 的 type 参数支持计数、概率密度和百分比 y 轴。图 14-35 显示了 "percent"（百分比）的风格：

```
histogram(~ Obama, obama_vs_mccain, type = "percent")
```

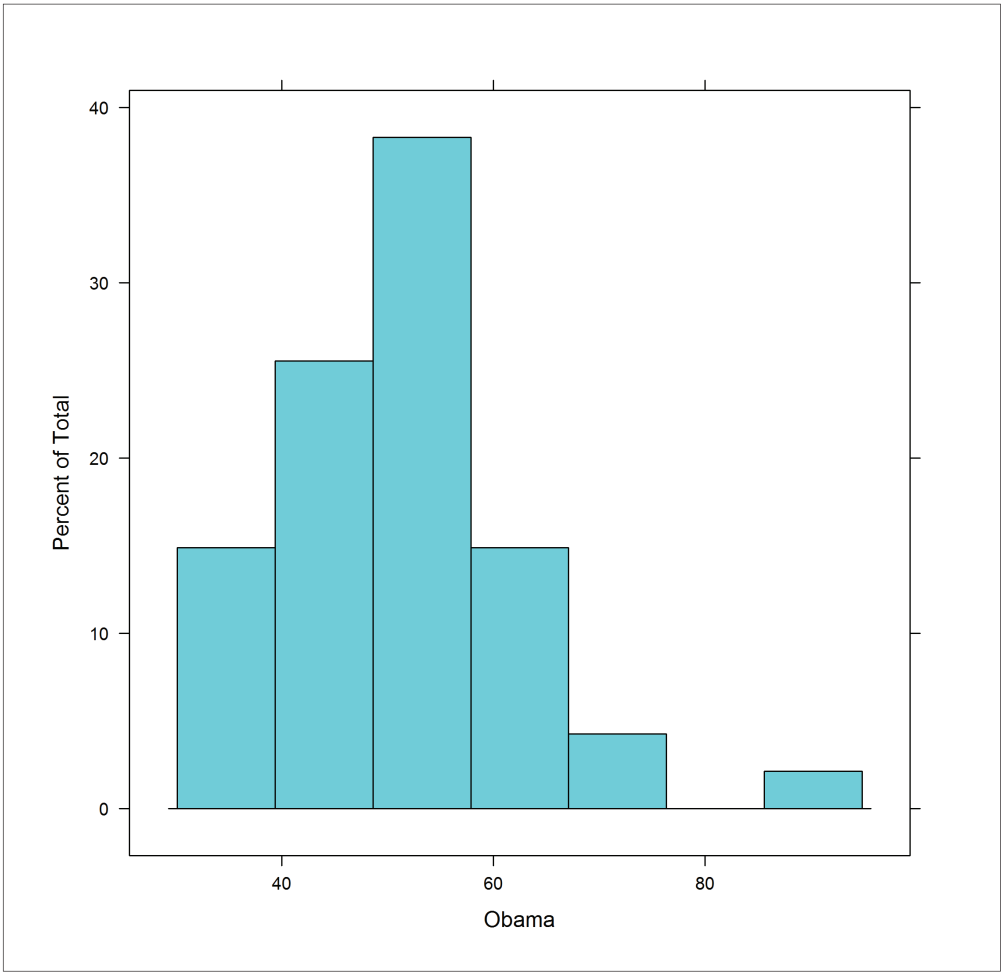


图 14-35：使用 `lattice` 系统绘制的按百分比缩放的直方图

`ggplot2` 直方图是通过添加一个直方图的 `geom` 来创建的。其区间规格很简单：只需要传递一个数字宽度到 `geom_histogram` 即可。其原因是为了迫使你手动尝试不同的区间数，而不仅仅满足于于默认值。图 14-36 显示了其用法：

```
ggplot(obama_vs_mccain, aes(Obama)) +
  geom_histogram(binwidth = 5)
```

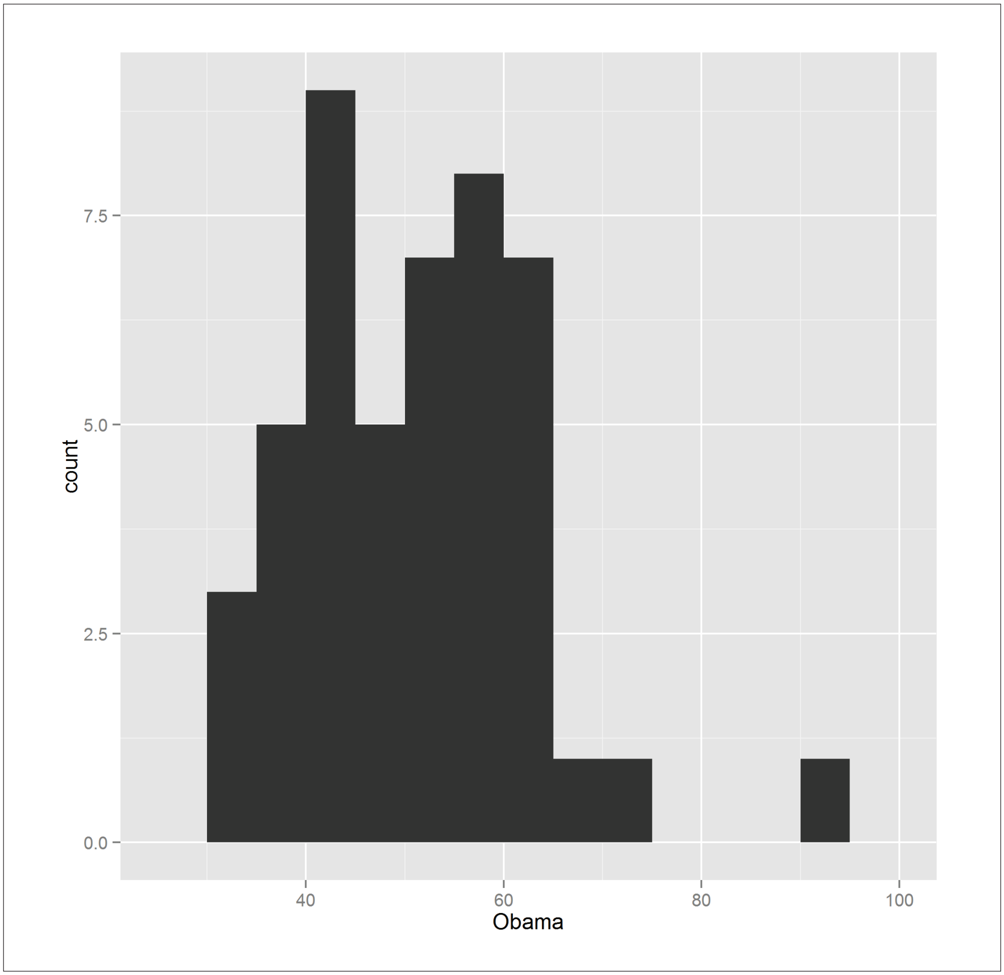


图 14-36：使用 ggplot2 系统绘制的直方图

你可以通过传递特殊的名字 ..count.. 或 ..density.. 到 y-aesthetic 中来选择计数或密度。图 14-37 演示了使用 density 绘制的图形：

```
ggplot(obama_vs_mccain, aes(Obama, ..density..)) +
  geom_histogram(binwidth = 5)
```

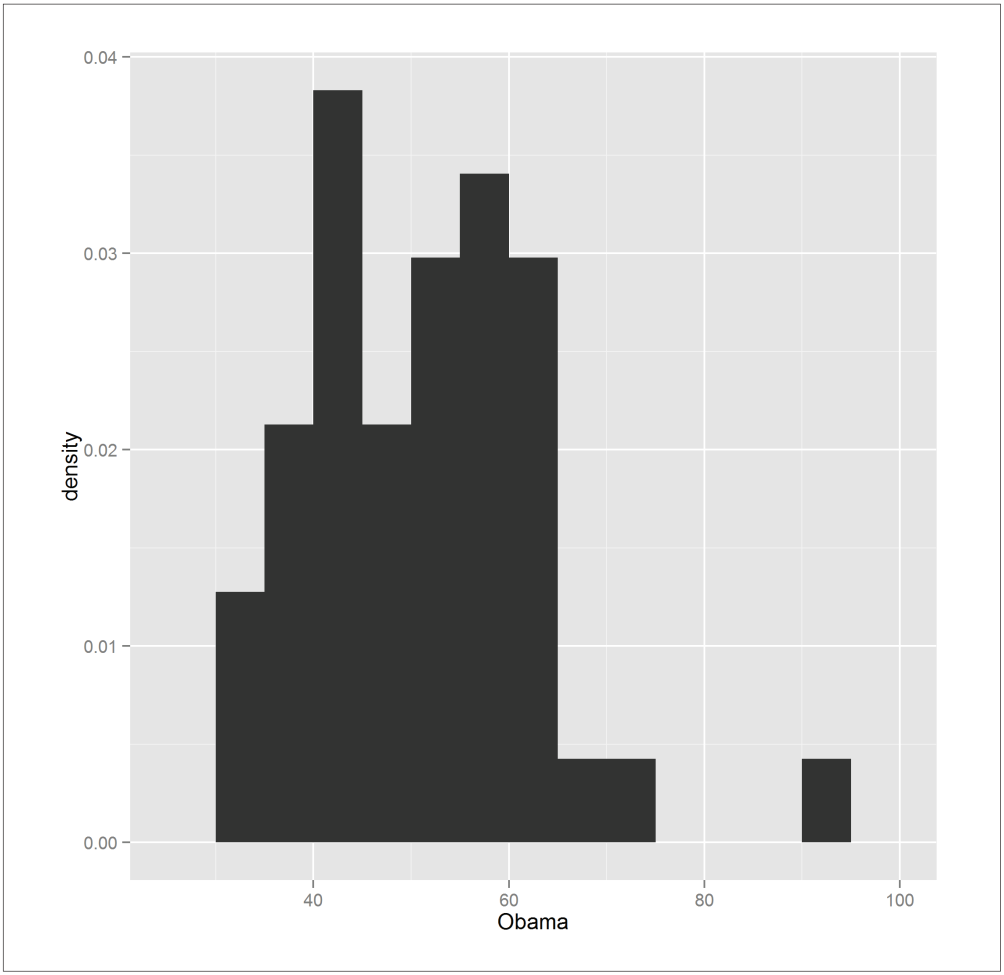


图 14-37：使用 ggplot2 系统绘制的概率密度直方图

14.7　箱线图

如果你想研究大量相关变量的分布，可绘制大量的直方图。例如，如果想按区域查看奥巴马得票数的分布，可以使用网格及切面法绘制 10 张直方图。这是可行的，但它不能很好地扩展。如果你需要一百张直方图，即使用上最大的显示器也放不下。箱线图（box plot，有时也被称为盒须图或盒形图）能节省大量空间，让你能轻松地一次比较众多的分布。尽管你不能得到像直方图或核密度图那样多的细节，但简单的高低和宽窄之间的比较是没问题的。

base 系统中绘制箱线图的函数为 boxplot。它在很大程度上受到 lattice 的启发——它使用一个公式接口且需要 data 参数。图 14-38 显示了其用法：

```
boxplot(Obama ~ Region, data = obama_vs_mccain)
```

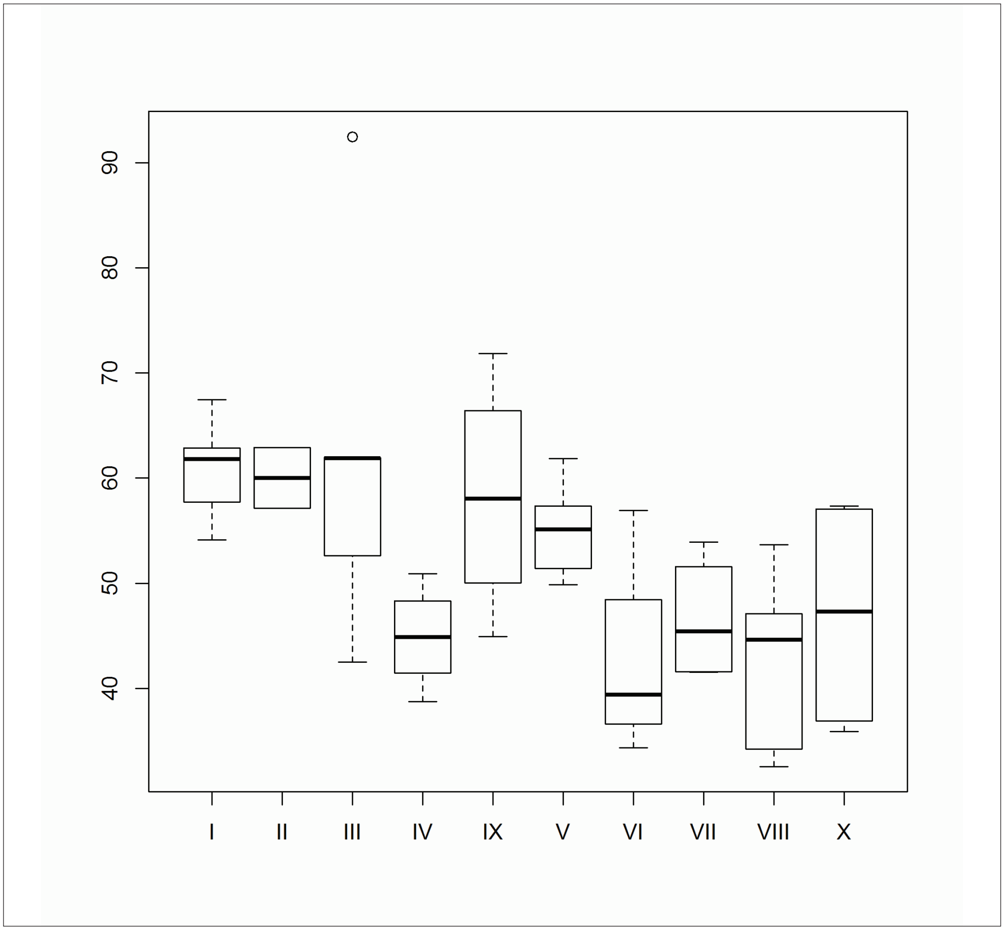


图 14-38：使用 base 系统绘制的箱线图

在某种意义上，如果我们重新把箱形图从小到大排列，这种类型的绘图往往会更清楚。reorder 函数能根据数值得分的情况改变因子水平的顺序。在图 14-39 中，我们按奥巴马得票的中位数给每个区域的 Region 级别打分：

```
ovm <- within(
  obama_vs_mccain,
  Region <- reorder(Region, Obama, median)
)
boxplot(Obama ~ Region, data = ovm)
```

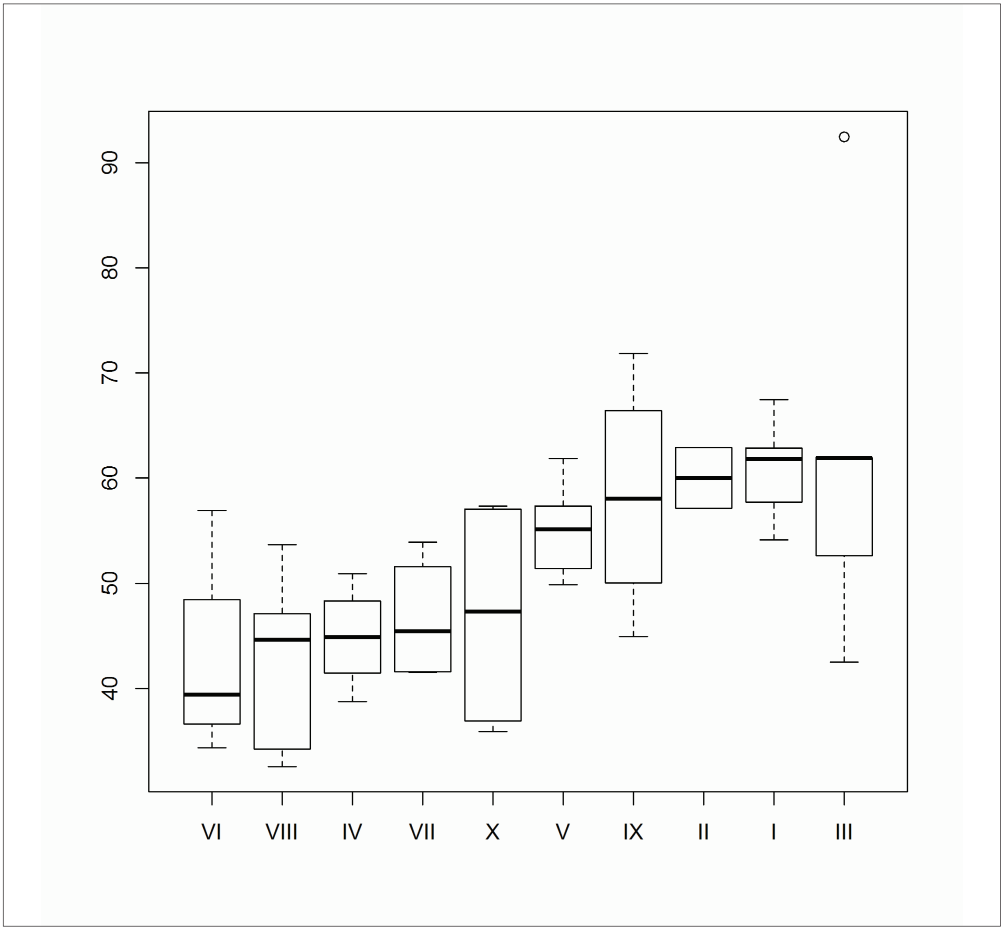


图 14-39：使用 base 系统为箱线图排序

从 base 切换到 lattice 非常简单。在本例中，我们可以直接将 boxplot 替换为 bwplot（bw 是 b（box）和 w（whisker）的简称）。请注意图 14-40 和图 14-38 的相似处：

```
bwplot(Obama ~ Region, data = ovm)
```

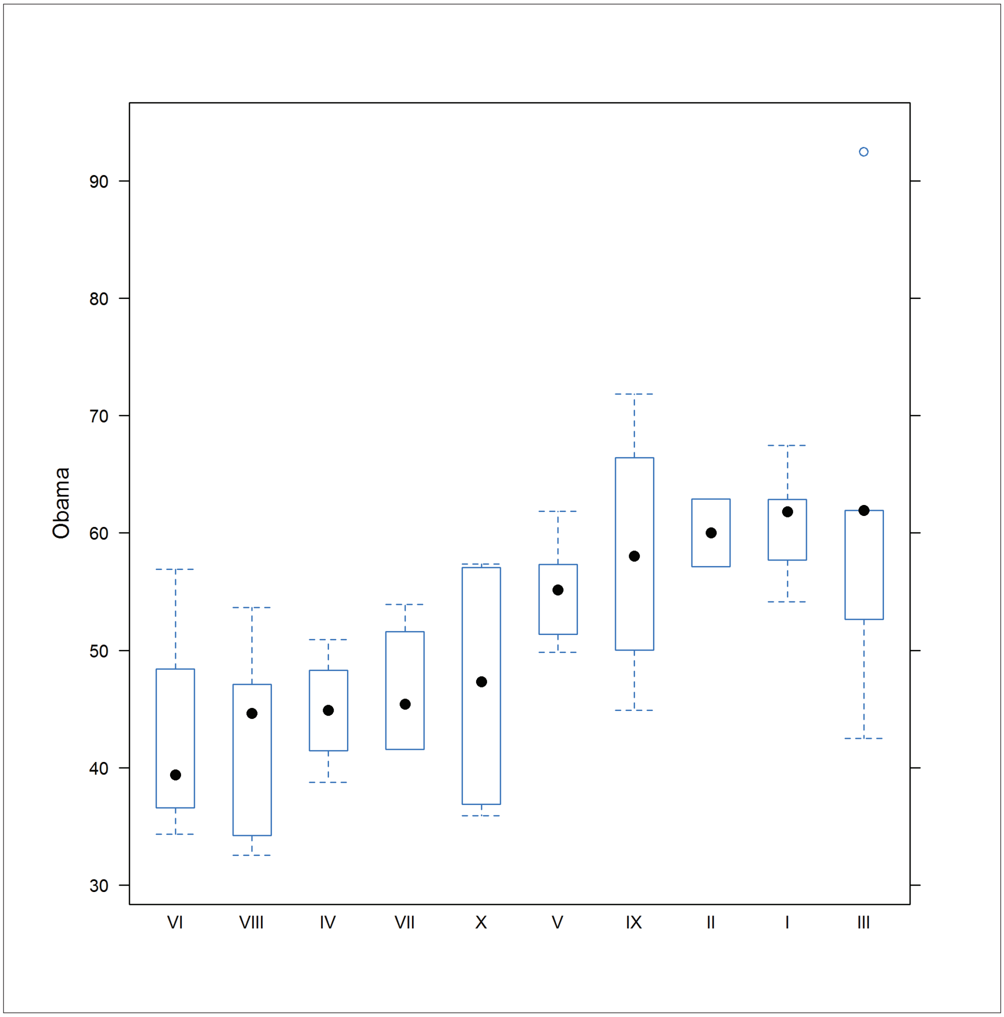


图 14-40：使用 lattice 绘制的箱线图

在 ggplot2 中绘制箱线图只需我们添加一个 geom_boxplot，如图 14-41 所示：

```
ggplot(ovm, aes(Region, Obama)) +
  geom_boxplot()
```

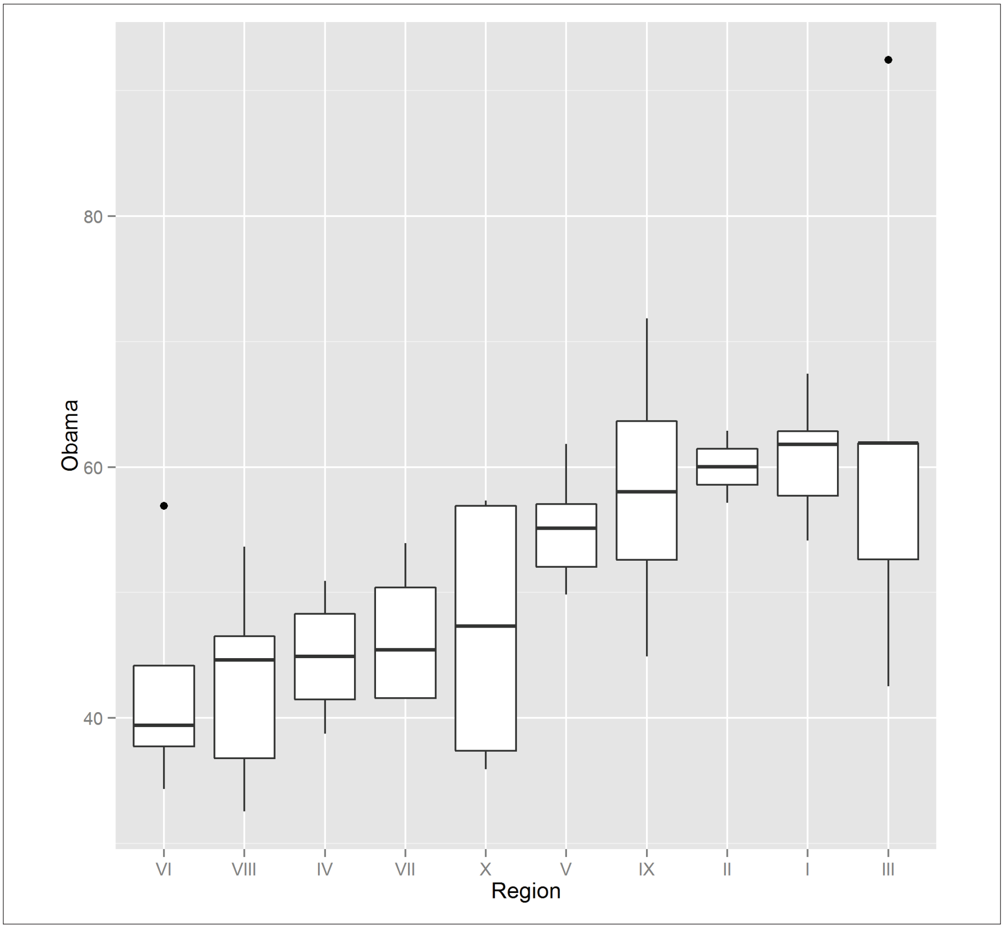


图 14-41：使用 ggplot2 系统绘制的箱线图

14.8　条形图

条形图（bar chart，又名柱状图、方框图等）是一种显示数值变量[7]的自然方式，此数值变量通常被类别变量所分割开。在下例中，我们来看看宗教身份在美国各州的分布情况。阿拉斯加和夏威夷的数据没有包括在数据集中，可以删除这些记录：

```
ovm <- ovm[!(ovm$State %in% c("Alaska", "Hawaii")), ]
```

注 7：更确切地说，它们必须是可计数的，或带有长度，又或其他能与零值相对比的值。对于对数比例来说，使用条形图其范围可能会延长到负无穷。在这种情况下，你应该使用点图。

在 base 系统中，条形图是使用 barplot 函数创建的。与 plot 函数一样，它也没有参数来指定一个数据框，所以我们需要把它置于 with 之内。barplot 的第一个参数包含条形的长度。如果这是一个命名向量（如果你没有出错，而且是从一个数据框里取得数据，这不会发生），那么这些名称将用作条形图上的标签。否则，正如这里的做法，你需要通过传递一个名为 names.arg 的参数指定标签值。默认情况下，条形都是垂直的。但为了使州名更可读，我们希望使用水平的条形图，这可以通过指定 horiz =TRUE 生成。

要显示各州的全名，我们还要通过 par 函数来调整一些绘图参数。由于历史的原因，大多数参数名称都只采用缩写，而不是更可读的值，这样代码看起来就会相当简洁。在修改 base 绘图之前，最好先看看 ?par 帮助页面。

las 参数（即 label axis style 的缩写）将控制标签是水平还是垂直的、是平行还是垂直于轴的。对于水平来说，如果你设置 las = 1，绘图通常会更具可读性。参数 mar 是一个长度为 4 的数值向量，它将分别给出绘图区下 / 左 / 上 / 右的边距宽度。我们真的很希望左手侧的边距宽一点，这样才能把州的名字都放进去。图 14-42 显示了以下代码的输出结果。

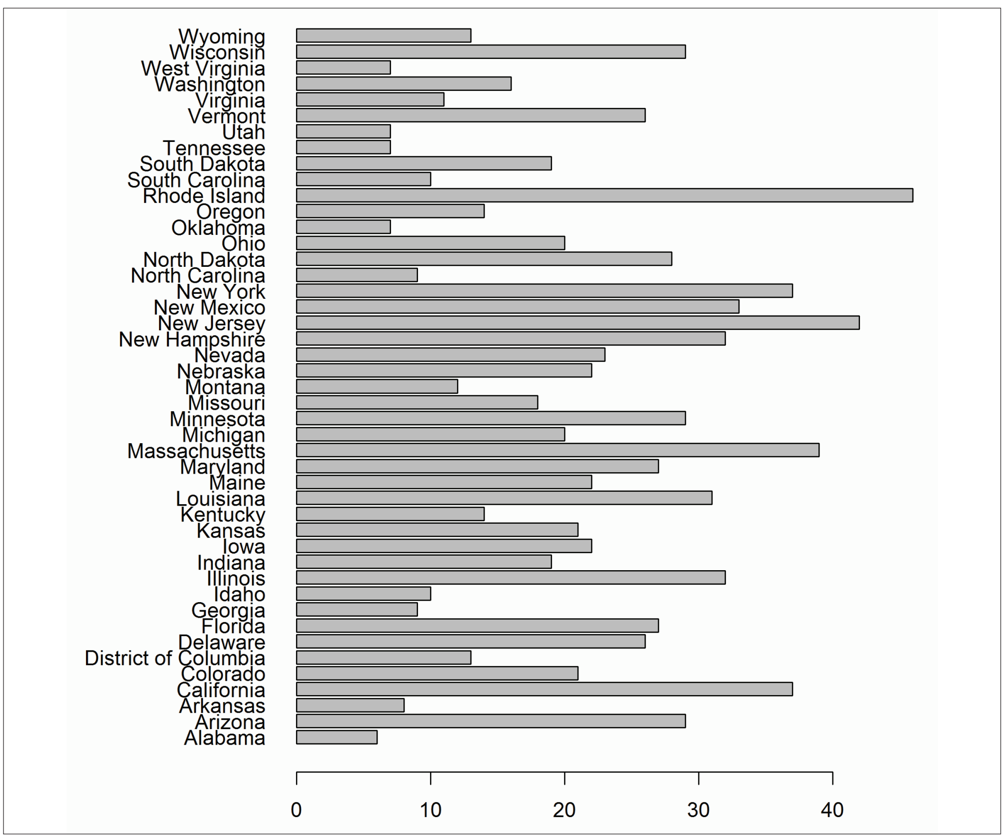


图 14-42：使用 base 系统绘制的条形图

```
par(las = 1, mar = c(3, 9, 1, 1))
with(ovm, barplot(Catholic, names.arg = State, horiz = TRUE))
```

像这样简单的条形图看上去挺好，但更有意思的是如何将含有几个变量的条形图放到一起。我们可以通过绘制天主教、新教、无宗教者以及其他栏来可视化宗教的分化情况。为了绘制多个变量，我们必须将它们逐行分别放入一个矩阵中（`rbind` 对此非常有用）。

这个矩阵的列名被用为条形的名字；如果没有列名，则须像上例一样使用 `names.arg` 来完成。默认情况下，图中每个变量的条形都彼此相邻，但由于我们正研究变量之间的分化情况，所以堆叠条形图更加合适。可以通过传递 `beside = FALSE` 参数实现这一点，如图 14-43 所示。

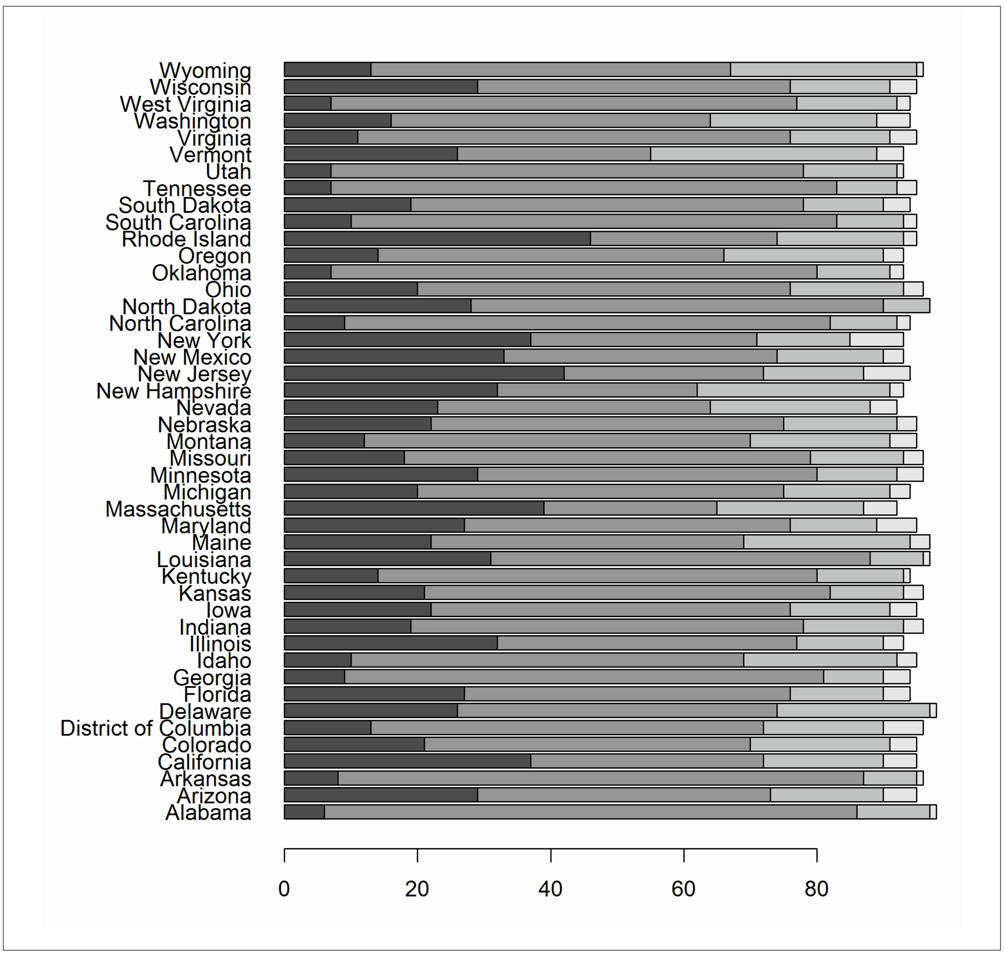


图 14-43：使用 base 系统绘制的堆积条形图

```
religions <- with(ovm, rbind(Catholic, Protestant, Non.religious, Other))
colnames(religions) <- ovm$State
par(las = 1, mar = c(3, 9, 1, 1))
barplot(religions, horiz = TRUE, beside = FALSE)
```

lattice 中与 barplot 等效的函数是 barchart，如图 14-44 所示。该公式接口与我们在散点图中所看到是一样的 yvar ~ XVAR：

```
barchart(State ~ Catholic, ovm)
```

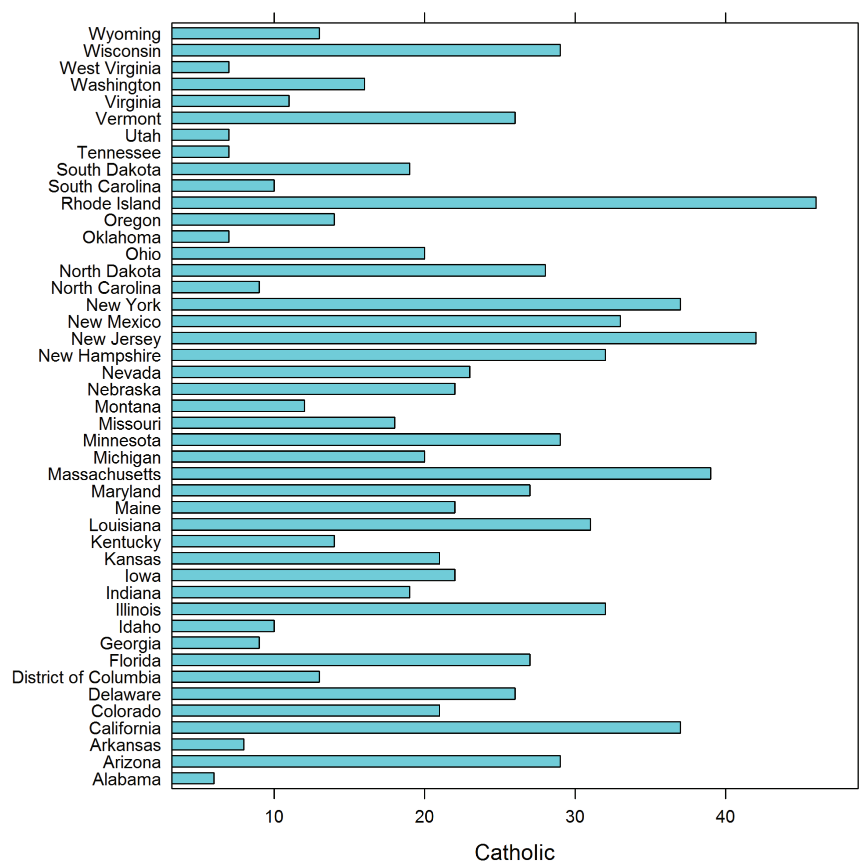


图 14-44：使用 lattice 系统绘制的条形图

把这个扩展到多个变量只需要调整一下公式，传递 stack = TRUE 即可绘制堆积的图形（见图 14-45）：

```
barchart(
  State ~ Catholic + Protestant + Non.religious + Other,
  ovm,
  stack = TRUE
)
```

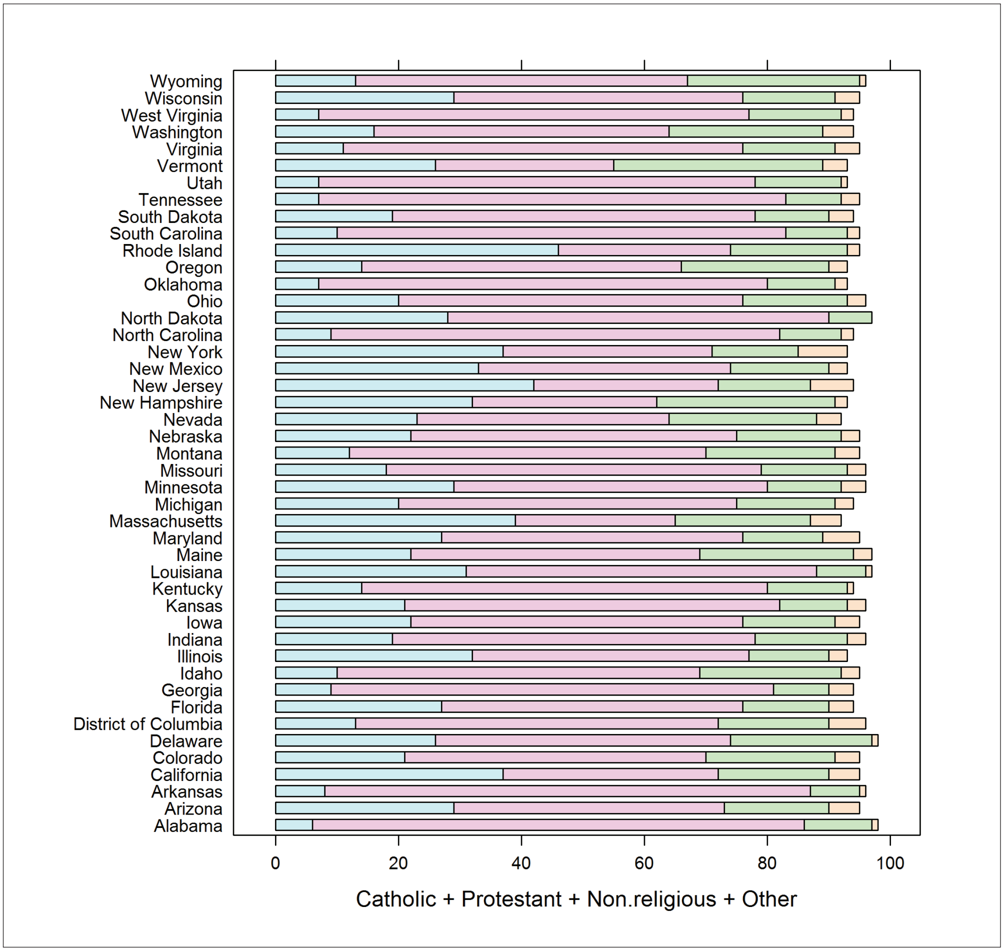


图 14-45：使用 lattice 系统绘制的堆积条形图

为了复制此图形，ggplot2 需要对数据作一些额外的工作。我们需要长表中的数据，所以必须首先对所需的列进行 melt 操作：

```
religions_long <- melt(
  ovm,
  id.vars = "State",
  measure.vars = c("Catholic", "Protestant", "Non.religious", "Other")
)
```

类似 base 系统，ggplot2 默认使用竖直的条形图；添加 coord_flip 可使它翻转为水平的条形图。最后，因为我们已知数据集中的每个条形的长度（无需再作计算），所以必须把 stat = "identity" 传递到 geom 中。默认情况下，条形图都是可堆积的，如图 14-46 所示：

```
ggplot(religions_long, aes(State, value, fill = variable)) +
  geom_bar(stat = "identity") +
  coord_flip()
```

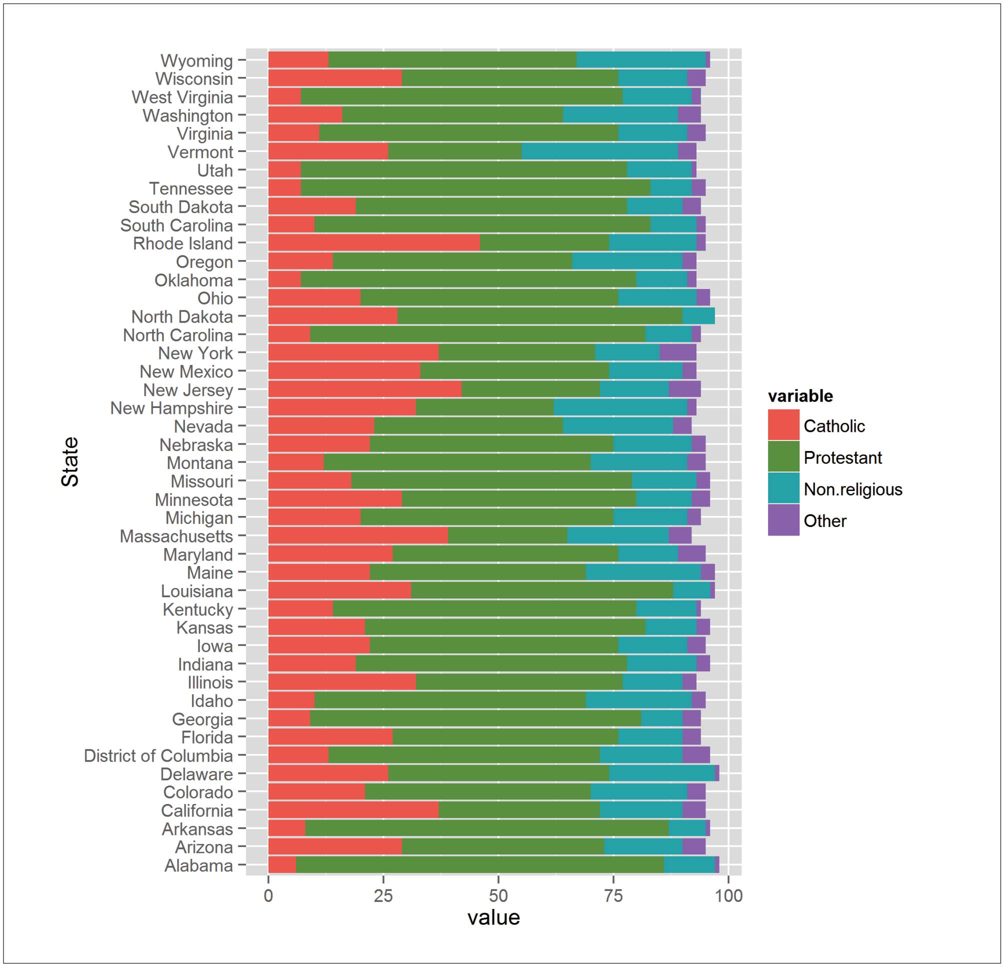


图 14-46：使用 ggplot2 绘制的堆积条形图

如果不想使用堆积，我们必须把参数 position = "dodge" 传递给 geom_bar。如图 14-47 所示：

```
ggplot(religions_long, aes(State, value, fill = variable)) +
  geom_bar(stat = "identity", position = "dodge") +
  coord_flip()
```

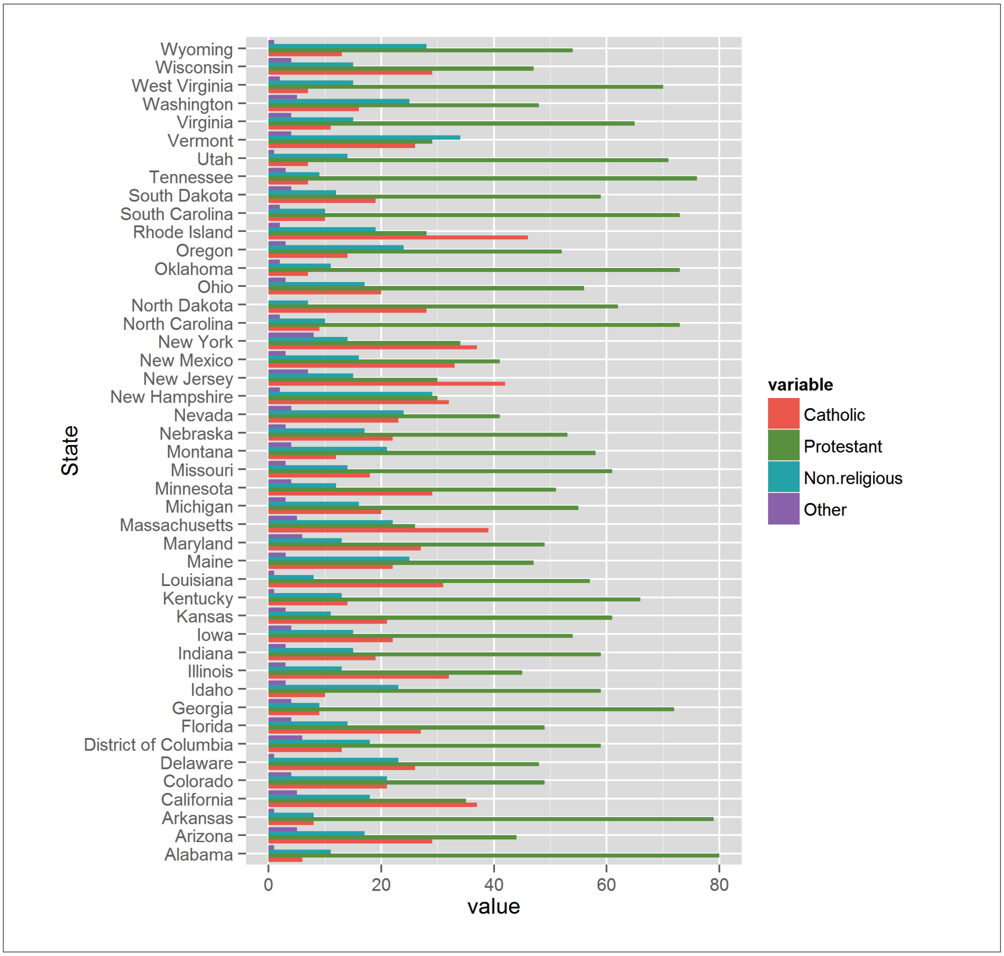


图 14-47：使用 ggplot2 绘制并列（dodged）条形图

此参数的另一种选项是 position = "fill"，它所创建出的每个堆积条都具有同样的高度，范围是 0~100% 。试试吧！

14.9　其他的绘图包和系统

还有许多其他的软件包具有绘图能力，它们都是基于上述三种图形系统中的一个或多个而实现的。例如，vcd 包有很多用于类别数据可视化的绘图函数（例如马赛克图和关联图），plotrix 有大量其他的绘图类型，还有一些特殊的图形分散在其他许多包中。

latticeExtra 和 GGally 扩展了 lattice 和 ggplot2 包，grid 提供了一种基础框架，能同时支持这两种系统。

你可能已经注意到，至今为止，所有已经涉及的图形都是静态的。其实已经有人多次尝试提供动态和交互式的图形[8]。 至今还没有十分完美的解决方案出现，但许多有意思的包正在做此尝试。

gridSVG 能让你把基于网格的图形（例如 lattice 或 ggplot2）输出到 SVG 文件中。这些是可以交互的，但它需要一些 JavaScript 的知识。playwith 允许点击操作与 base 或 lattice 系统互动。iplots 提供了整套额外的、更具交互式的绘图系统。这不太容易扩展，但是通用的绘图类型就在你面前，你可通过鼠标快速地研究数据。googleVis 提供了一个谷歌图表工具（Google Chart Tools）的封装包，它能创建在浏览器中显示的图形。rggobi 提供了一个 GGobi（用于可视化高维数据）的接口，以及 rgl 提供了一个 OpenGL 的交互式 3D 绘图接口。animation 包能让你生成 GIF 或 SWF 动画文件。

rCharts 包使用 lattice 语法封装了半打的 JavaScript 绘图库。它现在还不能通过 CRAN 取得，你需要从 GitHub 上安装它：

```
library(devtools)
install_github("rCharts", "ramnathv")
```

14.10 小结

- 我们可以计算大量的汇总统计。
- R 有 3 种图形系统：base、lattice 和 ggplot2。
- 每个系统都支持所有常见的绘图类型。
- R 中有一些函数能支持动态和交互式绘图。

14.11 知识测试：问题

- 问题 14-1
  min 和 pmin 函数之间的区别是什么？

- 问题 14-2
  在 base 图形系统中如何改变点的形状？

- 问题 14-3
  在 lattice 图形系统中，你将如何指定 x 和 y 变量？

注 8：动态的意思就是像动画一样；交互意味着可使用点击来改变它们。

- 问题 14-4
  什么是 `ggplot2` **aesthetic**？

- 问题 14-5
  如果要研究连续变量的分布，尽可能多地指出你能想到的图形类型的名称。

14.12 知识测试：练习

- 练习 14-1
  在 `obama_vs_mccain` 数据集中，找出各州中失业人员的百分比和人们投票支持奥巴马的百分比之间的（Pearson）相关性。[5]
  画出两个变量的散点图，使用你自己所选择的图形系统。（如果使用所有三个系统，则会有积分奖励）使用一个得 [10]，使用所有三个得。[30]

- 练习 14-2
  在 `alpe_d_huez2` 数据集中，绘制出车手最快时间的分布，把是否有（涉嫌）使用药物车手分开比较。使用 a）直方图和 b）箱线图显示它们。[10]

- 练习 14-3
  `gonorrhoea` 数据集包含了美国不同年份、年龄、种族和性别的淋病感染率。研究感染率随着年龄的增长如何变化。是否有一个时间趋势？种族和性别是否会影响感染率？[30]

第 15 章

分布与建模

使用汇总统计和图形能很好地帮助我们理解数据，但它们有一定的局限性。统计数据不能告诉你数据的形状，而图形不能扩展到多个变量（如果变量超过五六个就会变得很混乱）[1]，它们在数量上也不可扩展（你必须亲自察看每一个）。而且统计和图形都不擅长让你从数据中预测到什么。

这就是我们需要模型的原因：如果你已经充分理解了数据的结构并且能运行一个合适的模型，你就能通过对相关数据进行定量判断而作出预测。

目前已有大量的统计模型存在，还有更多的在不断地涌现，连大学的统计部门也难以消化。为了避免把这本书变成一门统计课，本章只处理一些非常简单的回归模型。如果你想另外了解统计学，建议阅读 *The R Book* 和 *Discovering Statistic Using R*，这两本书非常详细地解释了统计的概念。

在运行任何模型之前，我们需要一些关于如何生成随机数、各种分布和公式的背景知识。

15.1　本章目标

阅读本章后，你会了解以下内容：

- 如何生成符合多种分布的随机数；

注 1：如果你无意中已经使用过 biplot，那么非常好，你的确具有一些极客的天分。如果你使用一些诸如主成分分析或因子分析的降维技巧，绘制大量的变量是可行的，这些技巧能减少实际中所使用的变量数目。

- 如何从这些分布中找到位数和逆位数；
- 如何为一个模型编写公式；
- 如何运行、更新和绘制线性回归。

15.2 随机数

随机数对许多分析都至关重要，R 具有多种从不同分布中采样的函数。

15.2.1 示例函数

我们已多次看到 `sample` 函数（它首次出现在第 3 章）。这是一个用于生成随机数的重要的核心函数，而其行为有些怪诞，因而我们要对它做更深入的了解。如果你仅传递一个数值 `n` 给它，它将返回从一个 1 到 `n` 的自然数的排列：

```
sample(7)
```

```
## [1] 1 2 5 7 4 6 3
```

如果你给它传递第二个参数 `m`，它将返回 `m` 个 1 和 `n` 之间的随机数：

```
sample(7, 5)
```

```
## [1] 7 2 3 1 5
```

请注意，所有这些随机数都是不同的。默认情况下，`sample` 函数不会重复样本。也就是说，每个值只能出现一次。如果要允许有重复抽样，可传递 `replace = TRUE` 参数：

```
sample(7, 10, replace = TRUE)
```

```
## [1] 4 6 1 7 5 3 6 7 4 2
```

当然，大多数情况下，返回自然数并不有趣，但 `sample` 足够灵活，它可以让我们从任何喜欢的向量中采样。最常见的是给它传递一个字符向量：

```
sample(colors(), 5) # 为你家房子选择颜色的好方法
```

```
## [1] "grey53"      "deepskyblue2" "gray94"      "maroon2"
## [5] "gray18"
```

如果我们再冒些险，可以传递日期给它：

```
sample(.leap.seconds, 4)
```

```
## [1] "2012-07-01 01:00:00 BST" "1994-07-01 01:00:00 BST"
## [3] "1981-07-01 01:00:00 BST" "1990-01-01 00:00:00 GMT"
```

还可以通过传递 `prob` 参数来定义每个输入值的概率权重。在下例中，我们使用 R 来随机

地决定哪个月去度假，然后改变数据以增加我们放暑假的机会：

```
weights <- c(1, 1, 2, 3, 5, 8, 13, 21, 8, 3, 1, 1)
sample(month.abb, 1, prob = weights)

## [1] "Jul"
```

15.2.2 从分布中抽样

通常情况下，我们想根据某种概率分布来生成随机数。R 的各种包中一共提供了近百个分布函数、mixture 和系动词用于抽样。?Distribution 帮助页面文件记录了 R 基本包里的函数。如果你还想了解更多秘决，CRAN Task View: Probability Distributions（http://cran.r-project.org/web/views/Distributions.html）提供了更多的选择。

大部分的随机数生成函数的名称都是 r<distn>。例如，我们已经看到的 runif，它能生成均匀分布的随机数，还有 rnorm 能生成正态分布的随机数。这些函数的第一个参数都是要生成的随机数的数量，其太参数则会影响分布的形状。例如，runif 允许你设置分布的上限和下限：

```
runif(5)         # 生成 5 个介于 0 和 1 之间的均匀分布的随机数
runif(5, 1,10)   # 生成 5 个介于 0 和 10 之间的均匀分布的随机数
rnorm(5)         # 生成 5 个正态分布的随机数，它们的中位数为 0，标准差为 1
rnorm(5, 3, 7)   # 生成 5 个正态分布的随机数，它们的中位数为 3，标准差为 7
```

与任何其他软件一样，由 R 生成的随机数字实际上都是伪随机的。也就是说，它们是由某种算法而不是由真正的随机过程产生的。R 支持多种优质算法，就像 ?RNG 页面中所描述的一样。还有一些更专业的算法（如并行数生成）位于其他包中。以上提及的 CRAN Task View on distributions 也指出在哪里能找到更多这样的算法。你可以看到使用 RNGKind 函数以及哪些算法用于均匀和正态随机数的生成（从其他分布中采样通常也使用相同的均匀和正态随机数函数）：

```
RNGkind()

## [1] "Mersenne-Twister" "Inversion"
```

随机数发生器需要一个初始值来生成数字，此初始值即所谓的“种子”。通过把种子设置为特定的值，可保证每次运行同一段代码时能生成相同的随机数。例如，本书可以使用固定的种子值，使本书创建的范例在每次运行时都相同。使用 set.seed 函数设置种子值。它的输入参数为一个正整数。你选择哪个整数都无所谓，不同的种子将给出不同的随机数：

```
set.seed(1)
runif(5)

## [1] 0.2655 0.3721 0.5729 0.9082 0.2017
```

```
set.seed(1)
runif(5)

## [1] 0.2655 0.3721 0.5729 0.9082 0.2017

set.seed(1)
runif(5)

## [1] 0.2655 0.3721 0.5729 0.9082 0.2017
```

你也可以指定不同的生成算法，这种用法相当高级，不要贸然采用此做法除非很有把握。

15.3 分布

除了作为一个随机数生成器函数，大多数分布也有计算其概率密度函数（PDF）、累积分布函数（CDF）和 CDF 反函数的函数。

与 RNG 函数的名称以 `r` 开头类似，这些函数的名称分别以 `d`、`p` 和 `q` 开头。例如，正态分布具有 `dnorm`、`pnorm` 和 `qnorm` 函数。这些函数也许最好以图形来表示，如图 15-1 所示。(这里考虑到代码相当繁琐所以省略了。)

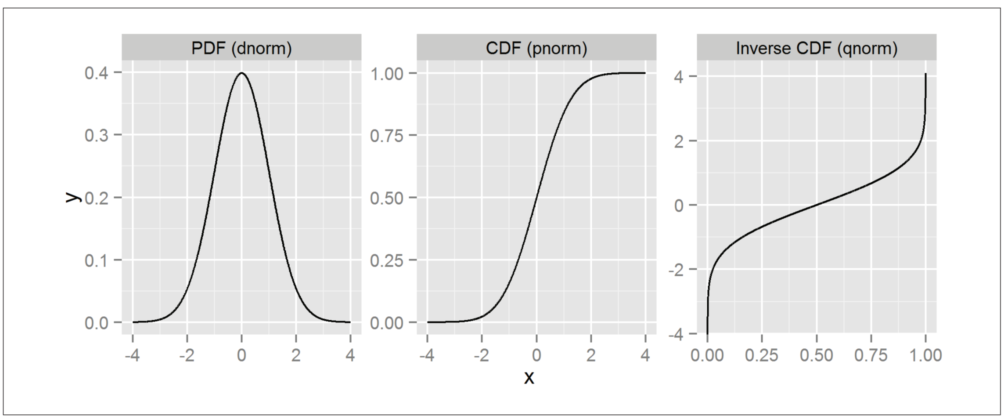


图 15-1：PDF、CDF 和正态分布的逆 CDF

15.4 公式

我们已经看到在 `lattice` 绘图和 `ggplot2` 面板中使用了公式。许多统计模型为了指定模型中变量的结构都类似地使用了公式。公式的确切含义取决于传递给它的模型函数（相同的公式分别出现在 `lattice` 绘图和 `ggplot2` 面板中时，它们的含义是不同的），但大多数模型都满足一个通用的模式：左边指定响应变量，右边指定自变量，两者由波浪线隔开。回到 `gonorrhoea` 数据集的例子，我们有：

```
Rate ~ Year + Age.Group + Ethnicity + Gender
```

在这里，`Rate` 是响应变量，我们选择了四个自变量（year/age group/ethnicity/gender）。对于可包括截距的模型（几乎都是回归模型），这样的公式隐式包含了截距。如果把它传递给一个线性回归模型，这将意味着：

$$Rate = \alpha_0 + \alpha_1 * Year + \alpha_2 * Age.Group + \alpha_3 * ethnicity + \alpha_4 * Gender + \epsilon$$

其中每个 α_i 是一个由该模型决定的常数，而 ϵ 是正态分布的误差项。

如果我们不想要截距项，可在右边添加一个零来取消它：

```
Rate ~ 0 + Year + Age.Group + Ethnicity + Gender
```

由该式给出的更新的公式为：

$$Rate = \alpha_1 * Year + \alpha_2 * Age.Group + \alpha_3 * ethnicity + \alpha_4 * Gender + \epsilon$$

每个这些公式仅包括自变量，它们之间没有任何交互。如果要包括交互，可以用星号代替加号：

```
Rate ~ Year * Age.Group * Ethnicity * Gender
```

这增加了所有可能的双向交互（year 和 age group，year 和 ethnicity 等），以及三方交互（year 和 age group 和 ethnicity 等），一直到所有自变量之间都有交互（year 和 age group 和 ethnicity 和 gender）。这往往就太多了，因为所有交互通常是没有意义的。

有两种方式能限制交互的程度。首先，你可以使用冒号新增个别的交互。在下例中，`Year: Ethnicity` 是这两者之间的交互，而 `Year: Ethnicity: Gender` 是一个三方的交互：

```
Rate ~ Year + Ethnicity + Gender + Year:Ethnicity + Year:Ethnicity:Gender
```

然而，如果你有超过三到四个变量，这种方法就会显得过于精细，会使输入变得很麻烦。使用 `^` 符号这种替代语法能让你包括所有的交互组合。下例包括了 `year`、`ethnicity` 和 `gender`，以及它们之间的三个双向交互：

```
Rate ~ (Year + Ethnicity + Gender) ^ 2
```

还可以包括变量的修饰版本。例如，环境进程生成了大量符合对数正态分布的变量，你可能希望把它们包含到一个 `log(var)` 的线性回归中。这些项可以直接包括在内，但你可能已经发现，包括 `var^2` 时会产生问题。因为该语法是保留给交互使用的，所以如果你想把幂计算包含在内，把它们包括在 `I()` 内吧：

```
Rate ~ I(Year ^ 2) # 是 Year 的平方，而不是交互
```

15.5 第一个模型：线性回归

普通最小二乘法线性回归在众多回归模型家族中是最简单的一个。用于计算它的函数是名字非常简约的 `lm`（即线性模型 linear model 的缩写）。它接受我们刚刚讨论过的公式类型，以及一个包含变量模型的数据框。来看一看 `gonorrhoea` 数据集。为简单起见，我们将忽略交互：

```
model1 <- lm(Rate ~ Year + Age.Group + Ethnicity + Gender, gonorrhoea)
```

如果我们打印出此模型变量，它将列出了每个输入变量的系数（α_i 的值）。如果仔细观察你会发现，对于我们放进模型的两个类别变量（age group 和 ethnicity），其中一个没有系数。例如，看不到 `0 to 4` 的年龄组，而 `American Indians & Alaskan Natives` 民族也不翼而飞。

这些"缺失"的类别被包含在截距中。在下面的输出中，截距中的 5540 人为 0~4 岁的美洲印第安人和阿拉斯加土著女性中每 10 万人在第 0 年感染淋病的数量。为了预测 2013 年的总感染率，加上 2013 乘以 `Year` 的系数 `-2.77`。为了预测在 25 到 29 岁同种族的感染率，加上该年龄组的系数 `291`：

```
model1

##
## Call:
## lm(formula = Rate ~ Year + Age.Group + Ethnicity + Gender, data = gonorrhoea)
##
## Coefficients:
##                      (Intercept)                               Year
##                         5540.496                             -2.770
##                   Age.Group5 to 9                 Age.Group10 to 14
##                           -0.614                             15.268
##                 Age.Group15 to 19                 Age.Group20 to 24
##                          415.698                            546.820
##                 Age.Group25 to 29                 Age.Group30 to 34
##                          291.098                            155.872
##                 Age.Group35 to 39                 Age.Group40 to 44
##                           84.612                             49.506
##                 Age.Group45 to 54                 Age.Group55 to 64
##                           27.364                              8.684
##               Age.Group65 or more  EthnicityAsians & Pacific Islanders
##                            1.178                            -82.923
##                EthnicityHispanics        EthnicityNon-Hispanic Blacks
##                          -49.000                            376.204
##      EthnicityNon-Hispanic Whites                        GenderMale
##                          -68.263                            -17.892
```

"0- to 4-year-old female American Indians and Alaskan Natives"（0 到 4 岁美洲印第安人和阿拉斯加土著女性）组被选中，是因为它包含了每个因子变量的第一级。我们可以通过遍历数据集和调用 `levels` 来查看这些因子水平：

```
lapply(Filter(is.factor, gonorrhoea), levels)

## $Age.Group
##  [1] "0 to 4"     "5 to 9"     "10 to 14"   "15 to 19"   "20 to 24"
##  [6] "25 to 29"   "30 to 34"   "35 to 39"   "40 to 44"   "45 to 54"
## [11] "55 to 64"   "65 or more"
##
## $Ethnicity
## [1] "American Indians & Alaskan Natives"
## [2] "Asians & Pacific Islanders"
## [3] "Hispanics"
## [4] "Non-Hispanic Blacks"
## [5] "Non-Hispanic Whites"
##
## $Gender
## [1] "Female" "Male"
```

除了知道每个输入变量的影响大小之外，我们通常还想知道哪些变量是最重要的。summary 函数被重载了，它可与 lm 一起工作来做到这一点。summary 输出中最令人兴奋的就是其系数表。Estimate 列显示了我们已看到过的系数，以及第四列 Pr(>|t|) 显示了 p 值。第五列给出了一个星级评定：所在 p 值小于 0.05 的变量得到一星，小于 0.01 为二星级，以此类推。这使得它便于快速查看哪些变量有显著效果：

```
summary(model1)


##
## Call:
## lm(formula = Rate ~ Year + Age.Group + Ethnicity + Gender, data = gonorrhoea)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -376.7 -130.6   37.1   90.7 1467.1
##
## Coefficients:
##                                     Estimate Std. Error t value Pr(>|t|)
## (Intercept)                         5540.496  14866.406    0.37   0.7095
## Year                                  -2.770      7.400   -0.37   0.7083
## Age.Group5 to 9                       -0.614     51.268   -0.01   0.9904
## Age.Group10 to 14                     15.268     51.268    0.30   0.7660
## Age.Group15 to 19                    415.698     51.268    8.11  3.0e-15
## Age.Group20 to 24                    546.820     51.268   10.67  < 2e-16
## Age.Group25 to 29                    291.098     51.268    5.68  2.2e-08
## Age.Group30 to 34                    155.872     51.268    3.04   0.0025
## Age.Group35 to 39                     84.612     51.268    1.65   0.0994
## Age.Group40 to 44                     49.506     51.268    0.97   0.3346
## Age.Group45 to 54                     27.364     51.268    0.53   0.5937
## Age.Group55 to 64                      8.684     51.268    0.17   0.8656
## Age.Group65 or more                    1.178     51.268    0.02   0.9817
## EthnicityAsians & Pacific Islanders  -82.923     33.093   -2.51   0.0125
## EthnicityHispanics                   -49.000     33.093   -1.48   0.1392
## EthnicityNon-Hispanic Blacks         376.204     33.093   11.37  < 2e-16
```

```
## EthnicityNon-Hispanic Whites          -68.263     33.093   -2.06   0.0396
## GenderMale                            -17.892     20.930   -0.85   0.3930
##
## (Intercept)
## Year
## Age.Group5 to 9
## Age.Group10 to 14
## Age.Group15 to 19                    ***
## Age.Group20 to 24                    ***
## Age.Group25 to 29                    ***
## Age.Group30 to 34                    **
## Age.Group35 to 39                    .
## Age.Group40 to 44
## Age.Group45 to 54
## Age.Group55 to 64
## Age.Group65 or more
## EthnicityAsians & Pacific Islanders *
## EthnicityHispanics
## EthnicityNon-Hispanic Blacks        ***
## EthnicityNon-Hispanic Whites        *
## GenderMale
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 256 on 582 degrees of freedom
## Multiple R-squared:  0.491, Adjusted R-squared:  0.476
## F-statistic: 33.1 on 17 and 582 DF,  p-value: <2e-16
```

15.5.1 比较和更新模型

通常，我们不会满足于所得到的第一个模型，而是想找到“最佳”模型或一些能提供洞见的模型。

本节展示了一些用于测量模型质量的指标，如 p 值和对数似然测量。通过使用这些指标来比较模型，你可以不断地自动更新模型，直到找到一个“最佳”模型为止。

不幸的是，像这样的模型自动更新（逐步回归）对于选择模型来说不是一个好的方法，而且当你增加了输入变量的数量，它会变得更糟。

更好的模型选择方法，如模型培训或模型平均，本书不作讨论。网站 CrossValidated statistics Q&A site（http://bit.ly/1a5q6IQ）列举了一些比较好方法。

> 试图找到“最佳”模型本身就是错的。好的模型即能对你的问题有所启发，而这样的模型可能有多个。

为了明智地给我们的数据选择一个模型，我们需要对影响淋病感染率的因素有所了解。淋病主要通过性传播（有一些是在母亲分娩的过程中传染给了婴儿），所以它最主要的驱动因素与性文化有关：如有多少个性伙伴，进行过多少次没有保护措施的性行为。

对于 Year 的 p 值是 0.71，这意味着它的影响还谈不上显著。在如此短的时间内（五年的数据），如果性文化上有任何变化能对感染率产生重要的影响，我会感到很惊讶，所以看看如果删除它会发生什么。

无须重新指定模型，我们可以使用 update 函数修改它。它接受一个模型和公式为输入参数。我们只更新公式的右边，左边保持不变。在下例中，. 的意思是：此项已在公式中，而减号 - 的意思是：删除此下一项：

```
model2 <- update(model1, ~ . - Year)
summary(model2)

##
## Call:
## lm(formula = Rate ~ Age.Group + Ethnicity + Gender, data = gonorrhoea)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -377.6 -128.4   34.6   92.2 1472.6
##
## Coefficients:
##                                     Estimate Std. Error t value Pr(>|t|)
## (Intercept)                          -25.103     43.116   -0.58   0.5606
## Age.Group5 to 9                       -0.614     51.230   -0.01   0.9904
## Age.Group10 to 14                     15.268     51.230    0.30   0.7658
## Age.Group15 to 19                    415.698     51.230    8.11  2.9e-15
## Age.Group20 to 24                    546.820     51.230   10.67  < 2e-16
## Age.Group25 to 29                    291.098     51.230    5.68  2.1e-08
## Age.Group30 to 34                    155.872     51.230    3.04   0.0025
## Age.Group35 to 39                     84.612     51.230    1.65   0.0992
## Age.Group40 to 44                     49.506     51.230    0.97   0.3343
## Age.Group45 to 54                     27.364     51.230    0.53   0.5934
## Age.Group55 to 64                      8.684     51.230    0.17   0.8655
## Age.Group65 or more                    1.178     51.230    0.02   0.9817
## EthnicityAsians & Pacific Islanders  -82.923     33.069   -2.51   0.0124
## EthnicityHispanics                   -49.000     33.069   -1.48   0.1389
## EthnicityNon-Hispanic Blacks         376.204     33.069   11.38  < 2e-16
## EthnicityNon-Hispanic Whites         -68.263     33.069   -2.06   0.0394
## GenderMale                           -17.892     20.915   -0.86   0.3926
##
## (Intercept)
## Age.Group5 to 9
## Age.Group10 to 14
## Age.Group15 to 19                   ***
## Age.Group20 to 24                   ***
## Age.Group25 to 29                   ***
## Age.Group30 to 34                   **
## Age.Group35 to 39                   .
## Age.Group40 to 44
## Age.Group45 to 54
## Age.Group55 to 64
## Age.Group65 or more
## EthnicityAsians & Pacific Islanders *
```

```
## EthnicityHispanics
## EthnicityNon-Hispanic Blacks        ***
## EthnicityNon-Hispanic Whites        *
## GenderMale
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 256 on 583 degrees of freedom
## Multiple R-squared:  0.491,       Adjusted R-squared:  0.477
## F-statistic: 35.2 on 16 and 583 DF,  p-value: <2e-16
```

anova 函数能计算模型的方差分析表（ANalysis Of VAriance），让你看到简化模型与全面的模型相比是否有显著区别：

```
anova(model1, model2)

## Analysis of Variance Table
##
## Model 1: Rate ~ Year + Age.Group + Ethnicity + Gender
## Model 2: Rate ~ Age.Group + Ethnicity + Gender
##   Res.Df      RSS Df Sum of Sq    F Pr(>F)
## 1    582 38243062
## 2    583 38252272 -1     -9210 0.14   0.71
```

右边列中的 p 值是 0.71，所以删除 year 一项并没有显著影响拟合此数据的模型。

赤池（Akaike）和贝叶斯（ Bayesian）信息准则提供了另外两种比较模型的方法，即 AIC 和 BIC 函数。它们利用了对数似然值，它们能告诉你用这个模型来拟合数据会有多好，而且会根据模型的项数多少决定如何作出惩罚（所以简单的模型比复杂的更好）。大致上较小的数字对应于“更好”的模型：

```
AIC(model1, model2)

##        df  AIC
## model1 19 8378
## model2 18 8376

BIC(model1, model2)

##        df  BIC
## model1 19 8462
## model2 18 8456
```

通过创建一个“不靠谱”的模型，你能更好地了解这些函数的影响。让我们删除 age group，它最能影响淋病感染率的预测（理应如此，儿童和老年人比青壮年的性行为要少得多）：

```
silly_model <- update(model1, ~ . - Age.Group)
anova(model1, silly_model)

## Analysis of Variance Table
##
```

```
## Model 1: Rate ~ Year + Age.Group + Ethnicity + Gender
## Model 2: Rate ~ Year + Ethnicity + Gender
##   Res.Df      RSS  Df Sum of Sq    F Pr(>F)
## 1    582 38243062
## 2    593 57212506 -11  -1.9e+07 26.2 <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

AIC(model1, silly_model)

##             df  AIC
## model1      19 8378
## silly_model  8 8598

BIC(model1, silly_model)

##             df  BIC
## model1      19 8462
## silly_model  8 8633
```

在这个“不靠谱”模型中，anova 指出模型间的显著差异，而且 AIC 和 BIC 都涨了不少。

继续寻找我们的“靠谱”模型，注意到性别是不显著的（p=0.39）。如果你做了练习 14-3（你已经完成了，是吧？），那么这对你来说可能是有趣的，因为从绘图中看起来妇女的感染率会更高。保留这种想法，直到你做完本章结尾的练习再看看。现在，让我们信任此 p 值，并从模型中删除性别 gender 项：

```
model3 <- update(model2, ~ . - Gender)
summary(model3)

##
## Call:
## lm(formula = Rate ~ Age.Group + Ethnicity, data = gonorrhoea)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -380.1 -136.1   35.8   87.4 1481.5
##
## Coefficients:
##                                      Estimate Std. Error t value Pr(>|t|)
## (Intercept)                           -34.050     41.820   -0.81   0.4159
## Age.Group5 to 9                        -0.614     51.218   -0.01   0.9904
## Age.Group10 to 14                      15.268     51.218    0.30   0.7657
## Age.Group15 to 19                     415.698     51.218    8.12  2.9e-15
## Age.Group20 to 24                     546.820     51.218   10.68  < 2e-16
## Age.Group25 to 29                     291.098     51.218    5.68  2.1e-08
## Age.Group30 to 34                     155.872     51.218    3.04   0.0024
## Age.Group35 to 39                      84.612     51.218    1.65   0.0991
## Age.Group40 to 44                      49.506     51.218    0.97   0.3342
## Age.Group45 to 54                      27.364     51.218    0.53   0.5934
## Age.Group55 to 64                       8.684     51.218    0.17   0.8654
## Age.Group65 or more                     1.178     51.218    0.02   0.9817
## EthnicityAsians & Pacific Islanders   -82.923     33.061   -2.51   0.0124
```

```
## EthnicityHispanics                  -49.000     33.061   -1.48   0.1389
## EthnicityNon-Hispanic Blacks        376.204     33.061   11.38  < 2e-16
## EthnicityNon-Hispanic Whites        -68.263     33.061   -2.06   0.0394
##
## (Intercept)
## Age.Group5 to 9
## Age.Group10 to 14
## Age.Group15 to 19                 ***
## Age.Group20 to 24                 ***
## Age.Group25 to 29                 ***
## Age.Group30 to 34                 **
## Age.Group35 to 39                 .
## Age.Group40 to 44
## Age.Group45 to 54
## Age.Group55 to 64
## Age.Group65 or more
## EthnicityAsians & Pacific Islanders *
## EthnicityHispanics
## EthnicityNon-Hispanic Blacks      ***
## EthnicityNon-Hispanic Whites      *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 256 on 584 degrees of freedom
## Multiple R-squared:  0.491,  Adjusted R-squared:  0.477
## F-statistic: 37.5 on 15 and 584 DF,  p-value: <2e-16
```

最后，截距项看起来是不显著的。这是因为默认组为 0~4 岁的美洲印第安人和阿拉斯加土著，他们没有太多的淋病患者。

可以使用 relevel 函数设置不同的默认值。作为一个在 30~34 岁的非西班牙裔白人[2]，我将有意把这个设定为默认值。在下例中，请注意，我们可以使用 update 函数来更新数据框及公式：

```
g2 <- within(
  gonorrhoea,
  {
    Age.Group <- relevel(Age.Group, "30 to 34")
    Ethnicity <- relevel(Ethnicity, "Non-Hispanic Whites")
  }
)
model4 <- update(model3, data = g2)
summary(model4)

##
## Call:
## lm(formula = Rate ~ Age.Group + Ethnicity, data = g2)
##
## Residuals:
##    Min     1Q Median     3Q    Max
```

注 2：请提醒我在第 2 版时更改此项！

```
## -380.1 -136.1   35.8   87.4 1481.5
##
## Coefficients:
##                                             Estimate Std. Error t value
## (Intercept)                                     53.6       41.8    1.28
## Age.Group0 to 4                               -155.9       51.2   -3.04
## Age.Group5 to 9                               -156.5       51.2   -3.06
## Age.Group10 to 14                             -140.6       51.2   -2.75
## Age.Group15 to 19                              259.8       51.2    5.07
## Age.Group20 to 24                              390.9       51.2    7.63
## Age.Group25 to 29                              135.2       51.2    2.64
## Age.Group35 to 39                              -71.3       51.2   -1.39
## Age.Group40 to 44                             -106.4       51.2   -2.08
## Age.Group45 to 54                             -128.5       51.2   -2.51
## Age.Group55 to 64                             -147.2       51.2   -2.87
## Age.Group65 or more                           -154.7       51.2   -3.02
## EthnicityAmerican Indians & Alaskan Natives     68.3       33.1    2.06
## EthnicityAsians & Pacific Islanders           -14.7       33.1   -0.44
## EthnicityHispanics                              19.3       33.1    0.58
## EthnicityNon-Hispanic Blacks                   444.5       33.1   13.44
##                                             Pr(>|t|)
## (Intercept)                                   0.2008
## Age.Group0 to 4                               0.0024 **
## Age.Group5 to 9                               0.0024 **
## Age.Group10 to 14                             0.0062 **
## Age.Group15 to 19                            5.3e-07 ***
## Age.Group20 to 24                            9.4e-14 ***
## Age.Group25 to 29                             0.0085 **
## Age.Group35 to 39                             0.1647
## Age.Group40 to 44                             0.0383 *
## Age.Group45 to 54                             0.0124 *
## Age.Group55 to 64                             0.0042 **
## Age.Group65 or more                           0.0026 **
## EthnicityAmerican Indians & Alaskan Natives   0.0394 *
## EthnicityAsians & Pacific Islanders          0.6576
## EthnicityHispanics                            0.5603
## EthnicityNon-Hispanic Blacks                 < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 256 on 584 degrees of freedom
## Multiple R-squared:  0.491, Adjusted R-squared:  0.477
## F-statistic: 37.5 on 15 and 584 DF,  p-value: <2e-16
```

因为现在参考点不同了，所以系数和 p 值也改变了，但还是有很多的星号出现在右侧的摘要输出，所以我们得知年龄和种族对感染率有影响。

15.5.2　绘图和模型检查

`lm` 模型有一个 `plot` 方法允许以 6 种不同的方式检查拟合度的好坏。在其最简单的形式中，你可以直接调用 `plot(the_model)`，它会逐张把图形绘制出来。稍好的方法是使用 `layout` 函数来一并查看所有的图，如图 15-2 所示 ：

```
plot_numbers <- 1:6
layout(matrix(plot_numbers, ncol = 2, byrow = TRUE))
plot(model4, plot_numbers)
```

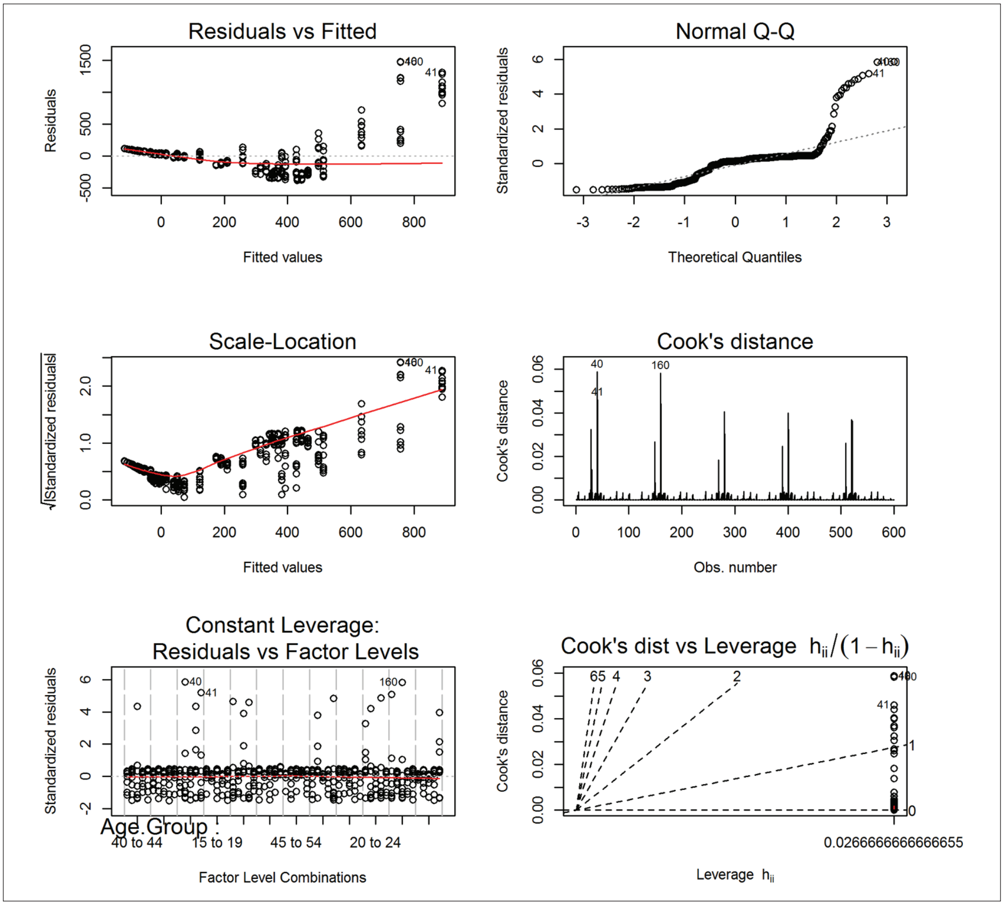


图 15-2：诊断图的线性模型

在最上面，有一些值过高（注意在“Residuals vs Fitted”绘图中右侧出现的大的正残差，那些点远高于“Normal Q-Q”绘图中的线，以及出现在“Cook’s distance”绘图中的峰值）。特别是第 40、41 和 160 行已被挑选出来作为异常值：

```
gonorrhoea[c(40, 41, 160), ]

##     Year Age.Group          Ethnicity Gender Rate
## 40  2007  15 to 19 Non-Hispanic Blacks Female 2239
## 41  2007  20 to 24 Non-Hispanic Blacks Female 2200
## 160 2008  15 to 19 Non-Hispanic Blacks Female 2233
```

所有这些大的值都指向非西班牙裔黑人女性，这表明或许我们错过了一个种族和性别的交互项。

由 lm 返回的模型变量相当复杂。为简便起见，在此我们不包括其输出，但你可以按照通常的方法来探索这些模型变量的结构：

```
str(model4)
unclass(model4)
```

有许多函数能很方便地访问模型中的各个组成部件，如 formula、nobs、residuals、fitted 和 coefficients：

```
formula(model4)

## Rate ~ Age.Group + Ethnicity
## <environment: 0x000000004ed4e110>

nobs(model4)

## [1] 600

head(residuals(model4))

##      1      2     3       4       5       6
## 102.61 102.93 87.25 -282.38 -367.61 -125.38

head(fitted(model4))

##       1       2      3      4      5      6
## -102.31 -102.93 -87.05 313.38 444.51 188.78

head(coefficients(model4))

##       (Intercept)  Age.Group0 to 4 Age.Group5 to 9 Age.Group10 to 14
##             53.56          -155.87         -156.49           -140.60
## Age.Group15 to 19 Age.Group20 to 24
##            259.83           390.95
```

除了这些，还有更多函数能用于诊断线性回归模型（在 ?influence.measures 页面中已列出）的质量，你也可从 summary 中取得 R^2 的值（由模型解释的方差部分）：

```
head(cooks.distance(model4))

##         1         2         3         4         5         6
## 0.0002824 0.0002842 0.0002042 0.0021390 0.0036250 0.0004217

summary(model4)$r.squared

## [1] 0.4906
```

这些辅助函数都能很好地替代诊断函数。举例来说，如果你不想使用 base 图形系统来绘制模型，可推出自己的 ggplot2 版本。图 15-3 显示残差与拟合值的一个绘图的例子：

```
diagnostics <- data.frame(
  residuals = residuals(model4),
```

```
  fitted = fitted(model4)
)
ggplot(diagnostics, aes(fitted, residuals)) +
  geom_point() +
  geom_smooth(method = "loess")
```

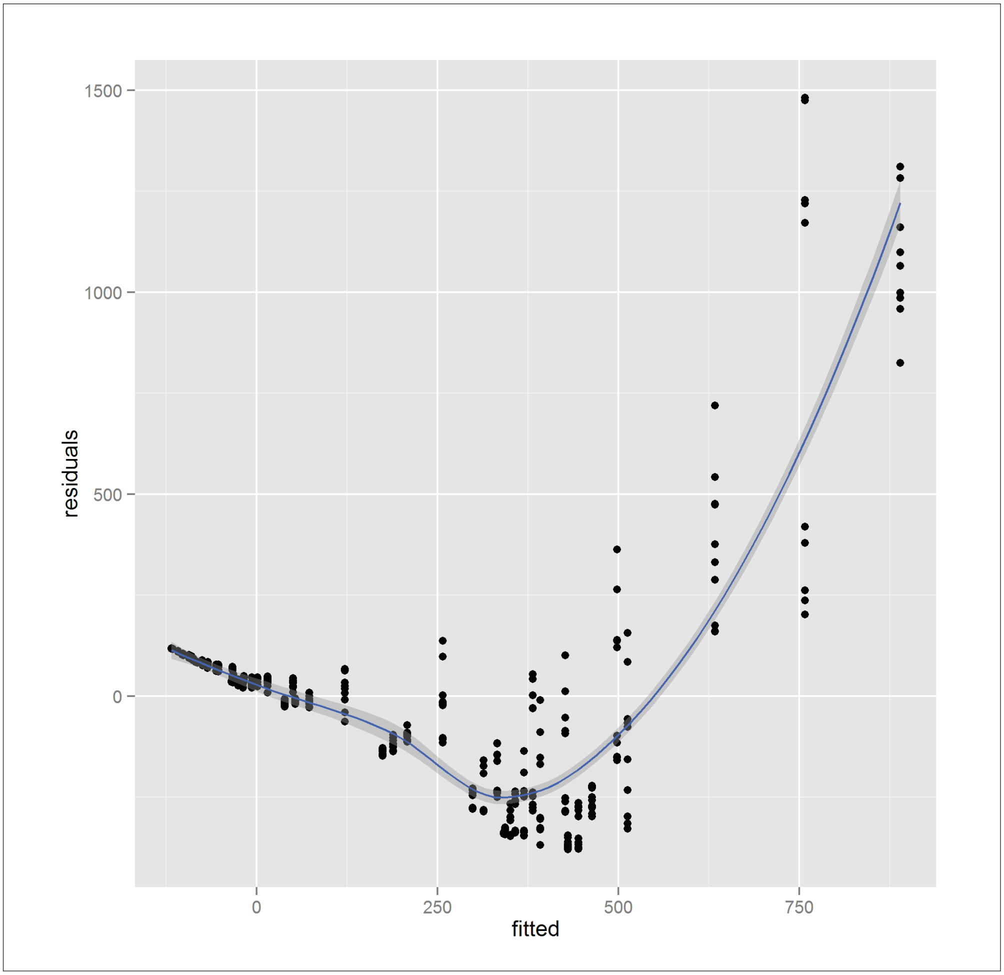


图 15-3：基于 ggplot2 的残差与拟合值的诊断图

使用模型的真正妙处在于，你可基于任何一种你感兴趣的模型输入来预测结果。更有趣的是，如果模型中有一个时间变量，你就可预测未来。在本例中，让我们来看一看特定人群的感染率。出于对自己所属人群情况的好奇，我想知道 30~34 岁的非西班牙裔白人的感染率：

```
new_data <- data.frame(
  Age.Group = "30 to 34",
  Ethnicity = "Non-Hispanic Whites"
)
```

```
predict(model4, new_data)

##     1
## 53.56
```

该模型预测的感染率为每 10 万人中会有 54 人。让我们把它与另一组的数据作对比：

```
subset(
  gonorrhoea,
  Age.Group == "30 to 34" & Ethnicity == "Non-Hispanic Whites"
)

##     Year Age.Group           Ethnicity  Gender Rate
## 7   2007  30 to 34 Non-Hispanic Whites    Male 41.0
## 19  2007  30 to 34 Non-Hispanic Whites Female  45.6
## 127 2008  30 to 34 Non-Hispanic Whites   Male  35.1
## 139 2008  30 to 34 Non-Hispanic Whites Female  40.8
## 247 2009  30 to 34 Non-Hispanic Whites   Male  34.6
## 259 2009  30 to 34 Non-Hispanic Whites Female  33.8
## 367 2010  30 to 34 Non-Hispanic Whites   Male  40.8
## 379 2010  30 to 34 Non-Hispanic Whites Female  38.5
## 487 2011  30 to 34 Non-Hispanic Whites   Male  48.0
## 499 2011  30 to 34 Non-Hispanic Whites Female  40.7
```

数据的范围为 34~48 ，所以我们的预测稍微有点高。

15.6 其他模型类型

线性回归仅是 R 建模能力的冰山一角。由于 R 现在是许多大学统计部门所选择的工具，所以几乎每个能想到的模型都会出现在 R 或它的某个包中。有很多书重点讲述如何使用 R 来完成统计方面的任务，本节仅简要指出哪里能够找到进一步的信息。

> 有许多人都编写了 R 的模型函数，它们分布在很多不同的包中，因此它们的语法各有千秋。`caret` 包的封装了大约 150 个模型，提供了一个一致的接口，里面还有用于模型训练和验证一些工具。Max Kuhn 的 *Applied Predictive Modeling* 是这个包的最佳指引。

线性（普通最小二乘回归）模型的 `lm` 函数有其泛化（generalization）版本函数 `glm`，此函数可让你为因变量上的误差项和变换指定不同的分布。你可将其用于 logistic 回归（其中的响应变量是逻辑或类别类型的）或其他，它的语法跟 `lm` 几乎完全相同：

```
glm(true_or_false ~ some + predictor + variables, data, family = binomial())
```

John Fox 的 *An R Companion to Applied Regression* 全书都在介绍用 `glm` 函数能完成哪些很酷的东西。

`nlme` 包（随 R 附带）包含了用于线性混合效应模型（linear mixed-effects model）的 `lme` 函数和用于非线性混合效应模型（nonlinear mixed-effects model）[3] 的 `nlme`。同样地，它的语法与 `lm` 或多或少都一样：

```
lme(y ~ some + fixed + effects, data, random = ~ 1 | random / effects)
```

José Pinheiro 和 Doug Bates 合著的 *Mixed-Effects Models in S and S-PLU* 是权威的参考书，其中有大量的范例。

对于成比例的响应变量，`betareg` 包中包含了一个同名函数，它允许进行 Beta 回归。这对于练习 15-3 尤其重要。

对于数据挖掘者（和其他涉及高维数据集的人士）来说，依附于 R 中的 `rpart` 包能创建回归树（又名决策树）。更令人兴奋的是，`randomForest` 包可以让你创建大量的回归树。`C50` 和 `mboost` 提供了梯度提升（gradient boosting），它在许多方面甚至比随机森林（random forrest）做得更好。

`kmeans` 可以让你使用 K 均值聚类（K-means clustering），还有几个包能提供专业扩展，例如 `kernlab` 为加权的 K 均值，`kml` 为纵向的 K 均值，`trimclust` 用于修剪的 K 均值，而 `skmeans` 为球形的 K 均值。

如果你做过大量的数据挖掘的工作，你可能会对 Rattle 很感兴趣，这是一个图形用户界面，它可以轻松地访问 R 的数据挖掘模型。Graham Willims 撰写了一本相关的书 *Data Mining With Rattle and R*。你可以先访问这个网站 http://rattle.togaware.com/，看看它是否吸引你。

社会科学家可能喜欢传统的降维模型（dimension-reduction model）：`factanal` 模型支持因子分析，而主成分分析有两种方法（`princomp` 能够兼容 S-Plus，而 `prcomp` 则采用了更现代的、数值更稳定的算法）。

`deSolve` 包中包含了很多能求解常 / 偏 / 延迟微分方程系统的方法。

15.7 小结

- 你能从几乎所有能想到的分布中生成相应的随机数。
- 大多数分布中有能计算它们的 PDF /CDF / 反 CDF 的函数。
- 许多模型函数使用公式来指定模型的形态。
- `lm` 能运行一个线性回归，且有许多辅助函数能更新、诊断和预测这些模型。
- R 能运行广泛的统计模型。

注 3：混合效应模型是一种回归，其中的一些要预测的变量会影响响应而不是平均值的方差。例如，如果要测量人们血液中的氧气含量，那么你可能想知道人与人之间的差别，以及对某人的测量值之间的差别，但你不在乎某个特定的人的氧气水平是较高还是较低。

15.8 知识测试：问题

- 问题 15-1
  如何才能创建两组相同的随机数？

- 问题 15-2
  PDF 、CDF 和反 CDF 函数的全称是什么？

- 问题 15-3
  在模型公式的上下文中，一个冒号 : 是什么意思？

- 问题 15-4
  你能使用哪些函数来比较线性回归模型？

- 问题 15-5
  如何确定线性模型解释的方差部分？

15.9 知识测试：练习

- 练习 15-1
  在我写这本书的时候，每个段落大概会出现三个错别字。使用泊松分布的 PDF 函数 `dpois` 来查看我在一个给定的段落中恰好出现三个错别字的概率。[5]
  一个 25 岁的健康女性在特定时间进行无避孕措施的性行为，则她每个月怀孕的机率为 25%。使用负二项分布的 CDF 函数 `pnbinom` 来计算她一年后怀孕的概率。[5]
  你需要 23 个人才能使他们中的两到三个在同一天生日的机率达到 50%。使用生日分布的逆 CDF 函数 `qbirthday` 来计算你需要多少人才会使他们在同一天生日的机率达到 90%。[5]

- 练习 15-2
  重新对 `gonorrhea` 数据集进行线性回归分析，只考虑 15-34 岁的人。观察显著的预测变量是否有所不同？ [15]
  附加分题：研究一下如果把交互项加入到模型中的结果是什么。[15]

- 练习 15-3
  安装并加载 `betareg` 包。通过包里的 `betareg` 函数，使用 beta 回归来研究 `obama_vs_mccain` 数据集。把 `Obama` 一列作为响应变量。
  为简单起见，去掉“District of Columbia”这个异常区间，不用考虑交互，且只包括一个民族（ethnicity）和宗教（religion）列。（种族和宗教列并非独立，因为它们代表了总体中的组成部分。）抛开政治上的理解，纯粹是为了更新模型，你可以信任其 p 值。
  提示：你需要从 0 到 1 重新调整 `Obama` 列的范围。[30]

第 16 章

程序设计

编写数据分析的代码并非易事。本章将讲述错误发生机制，以及如何能从开始就尽量避免错误。我们会从问题报出后可能收到的不同类型的反馈开始，一直到遇到错误。然后，再来看看应该如何处理抛出的错误，以及如何通过调试来消灭它们。单元测试框架则能让你尽量减少错误的代码。

接下来，我们将介绍一些魔术般的小技巧：如何将字符串转换成代码，以及将代码转换成字符串。（正如 Tommy Cooper 的口头禅："就像这样！"）本章最后将介绍一些 R 的面向对象编程系统。

16.1　本章目标

阅读本章后，你会了解以下内容：

- 如何通过消息、警告和错误给用户提供反馈；
- 如何优雅地处理这些错误；
- 一些调试代码的技巧；
- 如何使用 `RUnit` 和 `testthat` 单元测试框架；
- 如何将字符串转换为 R 的表达式及其相反的操作；
- S3 和参考类面向对象的编程系统的基础知识。

16.2　信息、警告和错误

我们已经在很多场合下见过 `print` 函数，它能把变量显示到控制台。R 有三种函数能把程

序状态的诊断信息显示出来。根据严重程度由小到大排序，它们分别是 message、warning 和 stop。

message 能把它所有的输入拼接起来，中间不需要任何的空格，然后把它们都写到控制台。它常用于对长时间运行的函数提供状态更新，或当要改变一个函数时把新的行为通知给用户，又或是提供默认参数的信息：

```
f <- function(x)
{
  message("'x' contains ", toString(x))
  x
}
f(letters[1:5])

## 'x' contains a, b, c, d, e
## [1] "a" "b" "c" "d" "e"
```

比起 print（或更低级的 cat），消息（message）函数的主要优点是用户可以关闭其显示。虽然这看似微不足道，但是当你要重复运行相同的代码时，就可以避免总是不断地看到同样的消息，令你倍受鼓舞：

```
suppressMessages(f(letters[1:5]))

## [1] "a" "b" "c" "d" "e"
```

警告（warning）的行为与消息非常相似，但它还有一些额外的特性能特别指出坏消息。警告适用于以下情景：当系统出了问题，但并非错得离谱以至于你的代码放弃执行。常见的例子是：糟糕的用户输入、较差的数值精度，或意想不到的副作用：

```
g <- function(x)
{
  if(any(x < 0))
  {
    warning("'x' contains negative values: ", toString(x[x < 0]))
  }
  x
}
g(c(3, -7, 2, -9))

## Warning: 'x' contains negative values: -7, -9
## [1] 3 -7 2 -9
```

和消息一样，警告可以被抑制：

```
suppressWarnings(g(c(3, -7, 2, -9)))

## [1] 3 -7 2 -9
```

有一个全局选项 warn，它确定如何处理警告。默认情况下 warn 取值为 0，这意味着只有当你的代码运行完毕，警告才会出现。

使用 `getOption` 可见 `warn` 选项的当前级别：

```
getOption("warn")

## [1] 1
```

如果该值小于零，所有的警告都将被忽略：

```
old_ops <- options(warn = -1)
g(c(3, -7, 2, -9))

## [1] 3 -7 2 -9
```

但通常来说，完全关闭警告是很危险的，所以你应该使用以下命令把选项重置为之前的状态：

```
options(old_ops)
```

把 `warn` 设置为 `1` 意味着只要发生了警告就马上显示它们，如果值为 `2` 或其他更大的值则意味着所有的警告都将变成错误。

可以通过输入 `last.warning` 访问最后发生的警告。

我们在前面曾提到，如果 `warn` 选项设置为 `0`，那么当你的代码运行完成后才会显示警告。其实这稍微有点复杂。如果只生成了 10 个或更少的警告，那么之前的说法完全正确。但是如果警告超过 10 个，你会得到一个消息，说明一共已经生成了多少次警告，而你也必须键入 `warnings()` 来查看它们。如图 16-1 所示。

```
> generate_n_warnings <- function(n) {
+ for (i in seq_len(n)) {
+ warning("This is warning ", i)
+ }
+ }
> generate_n_warnings(3)
Warning messages:
1: In generate_n_warnings(3) : This is warning 1
2: In generate_n_warnings(3) : This is warning 2
3: In generate_n_warnings(3) : This is warning 3
> generate_n_warnings(11)
There were 11 warnings (use warnings() to see them)
>
```

图 16-1：警告出现次数多于 10 个时，使用 warning 来查看它们

错误（error）是最严重的情况，且会停止程序的执行。错误应该只用于错误发生时，或者你知道错误将会发生时。常见的原因包括无法纠正的（例如，通过 `as.*` 函数）错误输入，无法读取或写入文件，或严重的数值误差：

```
h <- function(x, na.rm = FALSE)
{
```

```
  if(!na.rm && any(is.na(x)))
  {
    stop("'x' has missing values.")
  }
  x
}
h(c(1, NA))

## Error: 'x' has missing values.
```

如果任何传递给 stopifnot 的表达式被判定为假时，它会抛出一个错误。这里提供了一个简单的方法来检查程序的状态是否符合预期：

```
h <- function(x, na.rm = FALSE)
{
  if(!na.rm)
  {
    stopifnot(!any(is.na(x)))
  }
  x
}
h(c(1, NA))

## Error: !any(is.na(x)) is not TRUE
```

对于更广泛的人性化的测试，使用 assertive 包：

```
library(assertive)
h <- function(x, na.rm = FALSE)
{
  if(!na.rm)
  {
    assert_all_are_not_na(x)
  }
  x
}
h(c(1, NA))

## Error: x contains NAs.
```

16.3 错误处理

有些任务本身就是有风险的。文件或数据库的读取和写入是出了名的容易出错，因为你对文件系统、网络或数据库没有完全的控制权。其实，每当 R 与其他软件进行交互（如通过 rJava 访问 Java 代码，通过 R2WinBUGS 访问 WinBUGS，或任何其他数百个 R 可以连接的软件），都有引发错误的可能。

对于这些危险的任务[1]，你需要决定出现问题时如何处理。有时，当错误报出时，即使停

注 1：好吧，连接到一个文件并不那么严重，但它在编程中却属于高风险。

止执行程序也是没有用的。例如，如果你正在遍历文件并导入它们，如果这时导入失败，你不会希望仅仅停止执行而失去所有已成功导入的数据。

事实上，这点可以概括为：任何时候如果你的循环中正在运行一些有风险的事，你不会想在一次迭代失败后就放弃所有之前的进展。在下例中，我们尝试将列表中的每个元素转换成数据帧：

```
to_convert <- list(
  first = sapply(letters[1:5], charToRaw),
  second = polyroot(c(1, 0, 0, 0, 1)),
  third = list(x = 1:2, y = 3:5)
)
```

如果我们只是纯粹运行以上代码而不加任何保护，它会失败：

```
lapply(to_convert, as.data.frame)
## Error: arguments imply differing number of rows: 2, 3
```

糟糕！第三个元素因为长度不同而不能转换，而我们也失去了所有进展。

防止彻底失败的最简单的方法就是把容易出错的代码包括在 `try` 函数里：

```
result <- try(lapply(to_convert, as.data.frame))
```

现在，虽然错误会被打印到控制台上，但是代码不会停止执行（你可以通过传递 `silent = TRUE` 来抑制这种消息）。

如果传递给 `try` 函数的代码执行成功（没有抛出任何错误），那么 `result` 就是原来计算的结果，和平时一样。如果代码运行失败，那么 `result` 将是一个 `try-error` 类的对象。这就是说，当你写了一行包括 `try` 的代码，下一行看起来应该是这样：

```
if(inherits(result, "try-error"))
{
  # 用于错误处理的代码
} else
{
  # 正常执行的代码
}

## NULL
```

因为你每次都需要包括这些额外的代码，所以使用 `try` 函数的代码看上去有点丑陋。要使之更美观[2]则可使用 `tryCatch`。`tryCatch` 接收一个表达式来安全运行，正如 `try` 一样，且它还有内置的错误处理机制。

为了处理一个错误，把一个函数传递给一个叫 `error` 的参数。这个 `error` 参数接受一个错

注 2：不要低估代码美观的重要性。比起写代码，你会花更多的时间阅读它。

误（从技术上讲，它是类 simpleError 的对象），你可以任意操作、打印或忽略它。如果这听起来很复杂，请不要担心：它在实际中非常容易。在下例中，当发生错误时，我们会打印错误消息并返回一个空的数据框：

```
tryCatch(
  lapply(to_convert, as.data.frame),
  error = function(e)
  {
    message("An error was thrown: ", e$message)
    data.frame()
  }
)
## An error was thrown: arguments imply differing number of rows: 2, 3
## data frame with 0 columns and 0 rows
```

使用 tryCatch 的另一个技巧：你可以把一个表达式传递给一个命名为 finally 的参数，无论错误是否抛出它都会运行（就像在我们连接数据库时所看到的 on.exit 函数一样）。

尽管我们已经尝试了 try 和 tryCatch，但是还没有解决问题：在遍历时，即使抛出了一个错误，我们也能保存迭代成功的那部分结果。

为了实现这一目标，需把 try 或 tryCatch 置于循环内：

```
lapply(
  to_convert,
  function(x)
  {
    tryCatch(
      as.data.frame(x),
      error = function(e) NULL
    )
  }
)

## $first
##    x
## a 61
## b 62
## c 63
## d 64
## e 65
##
## $second
##                x
## 1  0.7071+0.7071i
## 2 -0.7071+0.7071i
## 3 -0.7071-0.7071i
## 4  0.7071-0.7071i
##
## $third
## NULL
```

因为这是一个公共代码片断，所以 plyr 包中有一个函数 tryapply，它正好能处理这种情况，且看上去更精简：

```
tryapply(to_convert, as.data.frame)

## $first
##    x
## a 61
## b 62
## c 63
## d 64
## e 65
##
## $second
##                  x
## 1  0.7071+0.7071i
## 2 -0.7071+0.7071i
## 3 -0.7071-0.7071i
## 4  0.7071-0.7071i
```

眼尖的观察者可能会发现，错误处理在这种情况下被直接移除了。

16.4 调试

所有不那么简单的软件都会有错误[3]。当问题发生后，你要能找到它们发生在哪里，并怀着希望去寻找一种能修正这些错误的方法。如果这些是你自己的代码，那么尤其如此。如果问题发生在一个简单脚本中，你通常可以访问所有的变量，因此找到问题并不难。

问题往往被深深地掩藏在多个函数嵌套调用中的某处。在这种情况下，你需要一个策略，能在调用栈的每一层检查程序的状态（“调用栈”是个行话，它就是函数的调用列表，从中可以看到你的代码在哪一层被调用）。

当错误发生时，traceback 函数能告诉你最后一个错误的发生位置。首先，让我们定义一些可能会发生错误的函数：

```
outer_fn <- function(x) inner_fn(x)
inner_fn <- function(x) exp(x)
```

现在，让我们使用一个错误的输入调用 outer_fn（然后再调用 inner_fn）：

```
outer_fn(list(1))
## Error: non-numeric argument to mathematical function
```

traceback 现在能告诉我们错误发生前所调用的函数（请看图 16-2）。

注 3：航天飞机中的软件以在 420 000 行代码中只包含一个错误而闻名，但是它采用非常正式的开发方法、代码评审以及广泛的测试，这些成本可不低。

```
> outer_fn <- function(x) inner_fn(x)
> inner_fn <- function(x) exp(x)
> outer_fn(list(1))
Error in exp(x) : non-numeric argument to mathematical function
> traceback()
2: inner_fn(x) at #1
1: outer_fn(list(1))
> |
```

图 16-2：使用 traceback 查看调用栈

一般来说，如果它不是一个明显的 bug，我们不知道调用栈在哪里出现问题。一个合理的策略是在抛出错误的函数的开始，如果需要，沿着栈往上追溯。要做到这一点，我们需要一种方法来在抛出错误的地方停止执行代码。方法之一是在错误点之前添加 browser 函数（因为使用了 traceback，所以我们知道错误发生之处）：

```
inner_fn <- function(x)
{
  browser() # 在这里停止执行
  exp(x)
}
```

browser 会在到达时暂停执行，这让我们有时间来检查程序。在大多数情况下，我们会调用 ls.str 来查看当前所有的变量值。在本例中，我们看到 x 是一个列表而不是一个数值向量，这导致了 exp 失败。

找出错误的另一种策略是设置全局的 error 选项。这种策略最适于错误位于其他人写的包里面时，因为在那里很难调用 browser。（你可以使用 fixInNamespace 函数在安装包里面改变函数，此变化会持续到你把 R 关闭为止。）

error 选项能接受不带参数的函数，并且会在错误抛出时被调用。举个简单的例子，我们可以将其设置为发生错误后打印一条消息，如图 16-3 所示。

```
> oh_dear <- function() message("Oh dear!")
> old_ops <- options(error = oh_dear)
> stop("I will break your program!")
Error: I will break your program!
Oh dear!
> |
```

图 16-3：覆盖全局 error 选项

尽管显示一条表示同情的消息算是对错误的一点安慰，但这对解决问题并没有什么帮助。更为有用的方法是 R 中自带的 recover 函数。在错误抛出后，recover 可以让你跳进调用栈中的任何函数（如图 16-4 所示）。

```
> outer_fn(list(1))
Error in exp(x) : non-numeric argument to mathematical function

Enter a frame number, or 0 to exit

1: outer_fn(list(1))
2: #1: inner_fn(x)

Selection: |
```

图 16-4：使用 error = recover 的调用栈

你也可以通过 debug 函数来逐行调试函数。不过，使用 inner 和 outer 这些单行函数有点无聊，所以我们会以另一种方式进行测试。包含在 learningr 包中的 buggy_count 是 plyr 包中的 count 函数的错误版本，当你给它传递一个因子时它会很隐晦地出错。在命令行下只需按下 Enter 键即可一直运行此函数，直到找到问题：

```
debug(buggy_count)
x <- factor(sample(c("male", "female"), 20, replace = TRUE))
buggy_count(x)
```

count（以及经过我们扩展的 buggy_count 函数）接受一个数据框或向量作为第一个参数。如果 df 的参数是一个向量，那么该函数会将它插入到数据框中。

图 16-5 显示了当我们到达这部分代码时会发生什么。当 df 是一个因子时，我们希望它被放在一个数据框里面。不幸的是，is.vector 对因子会返回 FALSE，步进也将被忽略。因子不是向量，因为它们除了名字外还拥有属性。代码中真正需要包含的（并且是在正确的 plyr 版本中）是调用 is.atomic，它对于因子和其他向量类型都会返回 TRUE，和 numeric 一样。

```
Browse[2]>
debug at #3: if (is.vector(df)) {
    df <- data.frame(x = df)
}
Browse[2]> is.vector(df)
[1] FALSE
Browse[2]> is.atomic(df)
[1] TRUE
```

图 16-5：调试 buggy_count 函数

要退出调试器只需在命令行键入 Q。因为 debug 函数的作用，调试器将在每一个函数被调用时启动。要关闭调试器，只需调用 undebug：

```
undebug(buggy_count)
```

可以使用 debugonce 作为替代方案，它只在函数第一次被调用时调用调试器 [4]。

注 4：Open Analytics 的 Tobias Verbeke 曾经打趣说："debugonce 函数太乐观了，我想如果有 debugtwice 可能会更好。"

16.5 测试

为了避免写出糟糕的充满 bug 的代码，测试非常重要。*单元测试*的概念就是一小块一小块地测试你的代码。在 R 中，这意味着在函数级别测试。（*系统或集成测试*是对整个软件的大规模的测试，但是比起数据分析，它更适用于应用程序的开发。）

每次更改一个函数时，你可能会破坏其他依赖于它的函数。这就是说每次改变一个函数时，你需要测试它可能带来的一切影响。手动尝试是不可能的，或者至少是非常耗时和乏味的，所以通常你不愿意这样做。因此，你需要把任务自动化。在 R 中，你有两种选择：

`RUnit` 拥有“xUnit”的语法，这意味着它非常类似于 Java 的 `JUnit`、.NET 的 NUnit、Python 的 `PyUnit`，以及其他一整套单元测试套件。如果你已使用过其他语言的单元测试工具，它就非常容易上手。

`testthat` 有自己的语法，以及一些其他的特性。特别是它的测试缓存功能使它对于大型项目来说速度飞快。

来测试一下在 6.3 节中初次了解函数时编写的 `hypotenuse` 函数。它使用了一个你能用纸和笔就能计算的简单算法[5]。该函数包含在 `learning` 包：

```
hypotenuse <- function(x, y)
{
  sqrt(x ^ 2 + y ^ 2)
}
```

16.5.1 RUnit

在 `RUnit` 中，每个测试都是一个没有输入的函数。每个测试都会使用包中的某个 `check*` 函数来将某些代码（在本例中即调用 `hypotenuse`）运行的实际结果与其预期值相比较。下例使用 `checkEqualsNumeric`，因为是在比较两个数字：

```
library(RUnit)
test.hypotenuse.3_4.returns_5 <- function()
{
  expected <- 5
  actual <- hypotenuse(3, 4)
  checkEqualsNumeric(expected, actual)
}
```

目前的测试没有统一的命名约定，但 `RUnit` 会默认查找以 `test` 为开头的函数。在这里使用命名约定旨在最大限度地把问题说清楚。测试需要类似 `test.name_of_function.description_of_inputs.returns_a_value` 的名称形式。

注 5：如果你对那句“用纸和笔来计算”感到畏惧。恭喜！你正在积极地成为一个 R 用户。

有时候，我们要确保一个函数能以正确的方式失败。例如，我们能测试：当没有提供输入时，`hypotenuse` 会失败：

```
test.hypotenuse.no_inputs.fails <- function()
{
  checkException(hypotenuse())
}
```

许多算法在输入很小或很大时会损失精度，所以最好要测试这些边界条件。R 中可以表示的最小和最大的正数值由内置的 `.Machine` 常数 `double.xmin` 和 `double.xmax` 给出：

```
.Machine$double.xmin

## [1] 2.225e-308

.Machine$double.xmax

## [1] 1.798e+308
```

对于小型和大型测试，我们会选择往这些限制值靠近。在小数值的情况下，我们需要手动地缩小试验的容差 `tolerance`。默认情况下，`checkEqualsNumeric` 对于实际测量结果在 `1e-8` 的范围内会认为测试通过（它使用绝对而不是相对误差）。我们把该值设为一个比输入小几个数量级的值，以确保测试按计划地失败：

```
test.hypotenuse.very_small_inputs.returns_small_positive <- function()
{
  expected <- sqrt(2) * 1e-300
  actual <- hypotenuse(1e-300, 1e-300)
  checkEqualsNumeric(expected, actual, tolerance = 1e-305)
}
test.hypotenuse.very_large_inputs.returns_large_finite <- function()
{
  expected <- sqrt(2) * 1e300
  actual <- hypotenuse(1e300, 1e300)
  checkEqualsNumeric(expected, actual)
}
```

可能的测试有无数个。例如，如果我们传入的是缺失值会发生什么？是 `NULL` 值、无限值、字符值、向量、矩阵或数据帧呢？又或者是我们期望在非欧几里得空间里得到答案？要进行彻底的测试需要你展开充分的想像力。释放你内在的童心，构想突破性测试吧。现在，在此打住。把所有的测试保存到一个文件中；`RUnit` 会默认查找所有以“runit”开头并以 `.R` 为扩展名的文件。这些测试可以在 `learningr` 包的 tests 目录中找到。

既然已有一些测试用例，让我们来运行它们。过程有两步。

首先，使用 `defineTestSuite` 定义一个测试套件。这个函数接受一个字符串输入作为命名（将在其输出中使用），以及一个包含了你所有的测试的路径参数。如果没有按标准的方式命名你的测试函数或文件，可以提供一个模式来识别它们：

```
test_dir <- system.file("tests", package = "learningr")
suite <- defineTestSuite("hypotenuse suite", test_dir)
```

第二步是使用 runTestSuite 来运行它们（已根据需要添加了额外的换行符，以适应本书的书写格式）：

```
runTestSuite(suite)

##
##
## Executing test function test.hypotenuse.3_4.returns_5 ...
## done successfully.
##
##
##
## Executing test function test.hypotenuse.no_inputs.fails ...
## done successfully.
##
##
##
## Executing test function
## test.hypotenuse.very_large_inputs.returns_large_finite ...
## Timing stopped at: 0 0 0 done successfully.
##
##
##
## Executing test function
## test.hypotenuse.very_small_inputs.returns_small_positive ...
## Timing stopped at: 0 0 0 done successfully.

## Number of test functions: 4
## Number of errors: 0
## Number of failures: 2
```

这将运行每个它能找到的测试并显示是否通过、失败或抛出了错误。在这种情况下，你可以看到小和大的输入测试失败。是什么出错了呢？

算法的问题在于我们对每个输入求了平方值。对大的数字求平方使得它比 R 能表示的最大的数（双精度）还要大，所以其结果为无穷大。对非常小的数字求平方会使它们更小，以致于 R 认为其值为零。（有更好的算法可以避免这个问题，请参阅 ?hypotenus 帮助页面中的链接，讨论在实际使用中更好的算法。）

RUnit 没有内置的 checkWarning 函数来测试警告。要测试一个警告是否已被抛出，我们需要一个技巧：把 warn 选项设置为 2，使警告变成错误，然后当测试函数退出时使用 on.exit 恢复其原始值。记得 on.exit 内的代码在函数退出时总会运行，不管它是否成功完成或抛出一个错误：

```
test.log.minus1.throws_warning <- function()
{
```

```
  old_ops <- options(warn = 2)  # 警告变成错误
  on.exit(old_ops)              # 回复原有行为
  checkException(log(-1))
}
```

16.5.2 testthat

虽然 testthat 具有不同的语法，但其原理几乎相同。主要的区别在于：不是每个测试都是一个函数，它是通过调用包中的某个 expect_* 函数。例如，expect_equal 相当于 Runit 的 checkEqualsNumeric 函数。翻译过来的测试（也在 learningr 包的 tests 目录中）看起来是这样：

```
library(testthat)
expect_equal(hypotenuse(3, 4), 5)
expect_error(hypotenuse())
expect_equal(hypotenuse(1e-300, 1e-300), sqrt(2) * 1e-300, tol = 1e-305)
expect_equal(hypotenuse(1e300, 1e300), sqrt(2) * 1e300)
```

为了运行它，我们需要调用 test_file 函数，其参数为包含测试用例的文件名；或调用 test_dir，其参数为测试用例所在的目录的路径。因为只有一个文件，所以使用 test_file：

```
filename <- system.file(
  "tests",
  "testthat_hypotenuse_tests.R",
  package = "learningr"
)
test_file(filename)

## ..12
##
## 1. Failure: (unknown) -----------------------------------------------------
## learningr::hypotenuse(1e-300, 1e-300) not equal to sqrt(2) * 1e-300
## Mean relative difference: 1
##
## 2. Failure: (unknown) -----------------------------------------------------
## learningr::hypotenuse(1e+300, 1e+300) not equal to sqrt(2) * 1e+300
## Mean relative difference: Inf
```

运行此测试有两种方法：一是使用 test_that 在命令行（或者说看似更像是复制和粘贴）测试代码，另一种是 test_package，在一个包中运行所有测试，这会使测试没有输出的函数更容易。

与 RUnit 不同，警告可以直接通过 expect_warning 测试：

```
expect_warning(log(-1))
```

16.6 魔法

我们使用文本编辑器所编写的源代码只不过是一堆字符串。当我们运行代码时，R 需要解读一下这些字符串再执行相应的操作。这个过程首先从把字符串变成若干的语言变量类型开始。有时我们可能要逆转此过程，即把语言变量转换为字符串。

这两种任务都是比较高级的话题，就像一种黑暗魔法。正如每一部电影中的魔法一样，如果使用前没有理解清楚，你最终会承受糟糕的、意料之外的结果。另一方面，这里有一些秘密武器，使用它们须谨慎且有见地。

16.6.1 将字符串转换成代码

每当你在命令行中输入一行代码时，R 会把这个字符串转换成它能理解的东西。以下是一个简单的反正切函数的调用：

```
atan(c(-Inf, -1, 0, 1, Inf))
```

```
## [1] -1.5708 -0.7854 0.0000 0.7854 1.5708
```

可以通过使用 `quote` 函数来慢镜头细看这行代码到底发生了什么。`quote` 接受一个像上面一行的函数作为输入参数，并将返回一个 `call` 类的对象，它代表一个尚未进行计算（unevaluated）的函数调用的对象：

```
(quoted_r_code <- quote(atan(c(-Inf, -1, 0, 1, Inf))))
```

```
## atan(c(-Inf, -1, 0, 1, Inf))
```

```
class(quoted_r_code)
```

```
## [1] "call"
```

接下来，R 将对此调用进行计算。可使用 `eval` 函数来模仿此步骤：

```
eval(quoted_r_code)
```

```
## [1] -1.5708 -0.7854 0.0000 0.7854 1.5708
```

一般情况下，为执行你键入的代码，R 会运行类似 `eval(quote(the stuff you typed at the command line))` 的函数。

为了更好地了解 `call` 类型，把它转换成一个列表：

```
as.list(quoted_r_code)
```

```
## [[1]]
## atan
##
```

```
## [[2]]
## c(-Inf, -1, 0, 1, Inf)
```

第一个元素是被调用的函数，其他元素都是我们要传递给它的参数。

记住重要的是：在 R 中几乎一切都是函数。这有点夸张，但像 + 的运算符、`switch`、`if`、`for` 的语言结构以及分配和索引都是函数：

```
vapply(
  list(`+`, `if`, `for`, `<-`, `[`, `[[`),
  is.function,
  logical(1)
)
```

```
## [1] TRUE TRUE TRUE TRUE TRUE TRUE
```

这样做的结果是，你在命令行键入的任何东西真的是函数调用，这是为什么此输入会变成 `call` 对象的原因。

我们兜了一个大圈说明：有时我们需接受 R 代码文本，然后让 R 来执行它。事实上，我们已经看到两个函数对于这种特殊情况完全就是这样做的：`assign` 接受一个字符串，并使用该名称将值指派给一个变量；与之相反，`get` 从一个字符串输入中获取一个变量。

除了分配和获取变量之外，我们偶尔可能还需要接受任意 R 的代码文本并从 R 中执行它。你可能已经注意到，当使用 `quote` 函数时，我们只是直接把 R 代码敲进去而没有把它括在括号内。如果输入的是一个字符串（例如一个长度为一的字符向量），问题会稍微不同：我们必须“解析”字符串。当然，这要通过 `parse` 函数来完成。

`parse` 返回一个 `expression` 对象而不是一个调用。不用紧张，其实 `expression` 基本上就是一个调用列表。

> 调用和表达式的本质是很深奥的，就像黑暗魔法一样。当你用 R 大玩“起死回生”之术时，我可不为接下来的“僵尸覆灭”负责。如果你对这些奥秘有兴趣，请阅读随 R 附带的 R Language Definition 手册中的第 6 章。

当以这种方式调用 `parse` 时，必须明确声名 `text` 参数：

```
parsed_r_code <- parse(text = "atan(c(-Inf, -1, 0, 1, Inf))")
class(parsed_r_code)
```

```
## [1] "expression"
```

使用 `eval` 来计算引用的 `R` 代码：

```
eval(parsed_r_code)
```

```
## [1] -1.5708 -0.7854 0.0000 0.7854 1.5708
```

这种与字符串混合计算的小技巧比较好用，但由此产生的代码通常是脆弱和难于调试的，使你的代码难以维护。这就是我上面提到“僵尸覆灭”的含义。

16.6.2 把代码转换成字符串

有时，我们要解决的问题正好相反，即把代码转换成为字符串。最常见的场景是为了把变量的名称传递给函数。base 包中的直方图绘图函数 hist 包含一个默认的标题，它能告诉你数据变量的名称：

```
random_numbers <- rt(1000, 2)
hist(random_numbers)
```

我们也可以自己重新实现这个功能，只需使用两个函数：substitute 和 deparse。substitue 接受一些代码并返回一个语言对象。这通常会是一个 call 对象，正如我们使用 quote 所创建的一样，但偶尔也会是一个 name 对象，它是一种能保存变量名的特殊类型。（不用担心其中的具体细节，本节被称为“魔法”并非偶然。）

下一步就是把这个语言对象转换为字符串，即所谓的 deparsing。当你检查函数的用户输入时，此技术能提供有用的错误信息。让我们来看看 deparse-substitue 组合在实际中的效果：

```
divider <- function(numerator, denominator)
{
  if(denominator == 0)
  {
    denominator_name <- deparse(substitute(denominator))
    warning("The denominator, ", sQuote(denominator_name), ", is zero.")
  }
  numerator / denominator
}
top <- 3
bottom <- 0
divider(top, bottom)

## Warning: The denominator, 'bottom', is zero.

## [1] Inf
```

substitute 在与 eval 一起使用时还有另一项技巧。给 eval 传递一个环境变量或数据框，你就可告诉 R 到哪里去查找要计算的表达式了。

举个简单的例子，我们可以用这一招来取得 hafu 数据集中 Gender 栏的水平值：

```
eval(substitute(levels(Gender)), hafu)

## [1] "F" "M"
```

这也恰恰是 `with` 函数的工作原理：

```
with(hafu, levels(Gender))

## [1] "F" "M"
```

事实上，有很多函数都使用了此技巧：`subset` 在好几个地方都用到它，`lattice` 图形系统以此技巧来解析公式。此技巧还有一些其他的变化，可参考 Thomas Lumley 的“Standard nonstandard evaluation rules”。

16.7 面向对象编程

到目前为止，我们看到的 R 程序大部分都是函数式编程风格的程序。也就是说函数是第一类对象，但是我们通常会使用一个数据分析脚本来逐行运行它们。

在某数情况下，使用面向对象编程（OOP）是非常有用的。这意味着数据和被允许操作的函数将被存储在类的内部。在管理大型复杂的程序时，这是一个很好的工具，而且特别适合于 GUI 的开发（在 R 或其他地方）。有关此主题的更多内容请参考 Michael Lawrence 的 *Programming Graphical User Interface in R* 一书。

R 有六种不同的面向对象的系统。不过不用担心，对于新项目，你只需要其中的两种。

R 中内建有三个系统 。

(1) S3 是一个轻量级的用于重载函数（例如，根据输入类型的不同调用不同版本的函数）的系统。
(2) S4 是一个功能完备的面向对象系统，但它非常笨重和难于调试。所以通常它只用于历史遗留代码中。
(3) 引用类是替代 S4 的现代系统。

还有其他三个系统可以在插件包里找到（但对于新代码，一般只使用引用类即可）。

(1) `proto` 是一个基于原型编程的轻量级的包装。
(2) `R.oo` 把 S3 扩展为一个完全成熟的面向对象的系统。
(3) `OOP` 是引用类的早期版本，现已不存在。

在许多面向对象的编程语言中，函数被称为*方法*。在 R 中，这两个词是可以互换的，但“方法”往往用于面向对象的上下文。

16.7.1 S3类

有时，我们需要一个函数根据不同的输入类型有不同的行为。一个典型的例子就是 `print` 函数，它为不同的变量给出了不同样式的输出。S3 让我们对不同类型的变量调用不同的函数，而不必记住每个变量的名字。

`print` 函数非常简单——其实只需要一行：

```
print

## function (x, ...)
## UseMethod("print")
## <bytecode: 0x0000000018fad228>
## <environment: namespace:base>
```

它接受一个输入参数 `x`（以及 `...`，其中的省略号是必要的），并调用 `UseMethod("print")`。`UseMethod` 检查 `x` 的类，并寻找另一个名为 `print.class_of_x` 的函数，如果能找到就调用它。如果找不到，它会尝试调用 `print.default`。

例如，如果我们想打印一个 `Date` 变量，只需键入：

```
today <- Sys.Date()
print(today)

## [1] "2013-07-17"
```

`print` 将调用与 `Date` 相关的函数 `print.Date`：

```
print.Date

## function (x, max = NULL, ...)
## {
##     if (is.null(max))
##         max <- getOption("max.print", 9999L)
##     if (max < length(x)) {
##         print(format(x[seq_len(max)]), max = max, ...)
##         cat(" [ reached getOption(\"max.print\") -- omitted",
##             length(x) - max, "entries ]\n")
##     }
##     else print(format(x), max = max, ...)
##     invisible(x)
## }
## <bytecode: 0x0000000006dc19f0>
## <environment: namespace:base>
```

在 `print.Date` 里，我们的日期先被转换为一个字符向量（通过 `format`），然后 `print` 才再次被调用。因为没有 `print.character` 函数存在，所以这时 `UseMethod` 把任务委托给 `print.default`，这样我们的日期字符串就出现在控制台中。

> 如果一个类的特定方法不能被发现，且没有默认的方法，那么将抛出错误。

你可以使用 `methods` 函数查看函数中所有可用的方法。`print` 函数拥有超过 100 个方法，所以在这里我们只显示了前几个：

```
head(methods(print))

## [1] "print.abbrev"  "print.acf"     "print.AES"
## [4] "print.anova"   "print.Anova"   "print.anova.loglm"

methods(mean)

## [1] mean.Date    mean.default   mean.difftime  mean.POSIXct  mean.POSIXlt
## [6] mean.times*  mean.yearmon*  mean.yearqtr*  mean.zoo*
##
## Non-visible functions are asterisked
```

> 如果你在函数名中使用了点号，例如 `data.frame`，那么在 S3 中的方法调用中就可能会混乱。例如，`print.data.frame` 的意思可能是一个输入参数为 `frame` 的 `print.data` 方法，而正确含义却是 `data.frame` 对象的 `print` 方法。因此，对于新的函数名，使用 `lower_under_case` 或 `lowerCamelCase` 是首选。

16.7.2 引用类

引用类比 S3 和 S4 更接近于经典的 OOP 系统，且对于使用过 C++ 的类及其衍生物的人来说这会比较直观。

> 一个类（class）是变量如何被构建的通用模板。对象（object）是类的一个特定实例。举例来说，`1:10` 是 `numeric` 类的一个对象。

`setRefClass` 函数创建一个类的模板。以 R 的术语来说，它就是*类生成器*（class generator）。在其他一些语言中，这会被称为一个*类工厂*（class factory）。

让我们尝试构建一个 2 维点类来作为例子。这样使用 `setRefClass`：

```
my_class_generator <- setRefClass(
   "MyClass",
   fields = list(
      # 在此定义数据变量
   ),
   methods = list(
      # 在此定义操作数据的函数
      initialize = function(...)
```

```
          {
          # 初始化是一个特殊的函数
          # 它在对象被创建时调用。
          }
      )
  )
```

我们的类需要 x 和 y 坐标来存储它的位置，希望它们都是数值类型（numeric）。

在下例中，我们将 x 和 y 声明为数字：

> 如果不关心 x 和 y 的类型，则把它们声明为特殊值 ANY。

```
point_generator <- setRefClass(
  "point",
  fields = list(
    x = "numeric",
    y = "numeric"
  ),
  methods = list(
    #TODO
  )
)
```

这意味着，如果我们试图为它们分配另一种类型的值就会抛出错误。故意限制用户的输入可能听上去有些反常，但它能把你从之后可能出现的各种莫名其妙的错误中拯救出来。

接下来，我们需要添加一个 initialize 方法。在每次创建 point 对象时，它都会被调用。此方法接受 x 和 y 作为输入的数值，并将它们分配给 x 和 y 字段。有三种比较有趣的事情需要注意。

(1) 如果方法的第一行是一个字符串，那么它会被认为是此方法的帮助文本。
(2) 全局赋值运算符 <<- 用于字段赋值。本地分配符（使用 <-）只不过是在方法内创建局部变量。
(3) 最好的做法是不要让 initialize 函数接受任何参数，因为这会使各继承更容易一些，我们将在稍后看到这一点。这就是为什么 x 和 y 参数会有默认值的原因[6]。

通过使用 initialize 方法，我们的类生成器现在看起来是这样：

```
point_generator <- setRefClass(
  "point",
  fields = list(
    x = "numeric",
```

注 6：如果你仍然感到困惑，NA_real_ 是一个缺失值。通常对于缺失值来说我们只用 NA 以及让 R 来判断它所需要的类型，但在这种情况下，因为指定的字段必须是数字，所以我们必须明确地说明其类型。

```
    y = "numeric"
  ),
  methods = list(
    initialize = function(x = NA_real_, y = NA_real_)
    {
      "Assign x and y upon object creation."
      x <<- x
      y <<- y
    }
  )
)
```

我们点类生成器已经完成，所以我们现在可以创建一个 point 对象。每个生成器都有一个用于此目的的 new 方法。作为对象创建流程的一部分，new 方法将调用 initialize（如果存在的话）：

```
(a_point <- point_generator$new(5, 3))

## Reference class object of class "point"
## Field "x":
## [1] 5
## Field "y":
## [1] 3
```

生成器还有一个能返回你所指定的帮助字符串的 help 方法：

```
point_generator$help("initialize")

## Call:
## $initialize(x = , y = )
##
##
## Assign x and y upon object creation.
```

你也可以通过把类的方法包装在某个其他的函数里，这样就为面向对象的代码提供一个传统的接口。当你把代码发布给他人，又不想教其 OOP 时，这种方法比较有用：

```
create_point <- function(x, y)
{
   point_generator$new(x, y)
}
```

目前，此类不太有趣，因为它还不会做任何事情。让我们重新为它定义一些更多的方法：

```
point_generator <- setRefClass(
  "point",
  fields = list(
    x = "numeric",
    y = "numeric"
  ),
  methods = list(
                      initialize = function(x = NA_real_, y = NA_real_)
```

```
      {
        "Assign x and y upon object creation."
        x <<- x
        y <<- y
      },
      distanceFromOrigin = function()
      {
        "Euclidean distance from the origin"
        sqrt(x ^ 2 + y ^ 2)
    },
    add                   = function(point)
    {
        "Add another point to this point"
        x <<- x + point$x
        y <<- y + point$y
        .self
      }
    )
  )
```

这些额外的方法属于 point 对象，与属于类生成器的 new 和 help 不同（在面向对象编程的术语中，new 和 help 是静态方法）：

```
a_point <- create_point(3, 4)
a_point$distanceFromOrigin()

## [1] 5

another_point <- create_point(4, 2)
(a_point$add(another_point))

## Reference class object of class "point"
## Field "x":
## [1] 7
## Field "y":
## [1] 6
```

除了 new 和 help，生成器类还有几个其他的方法。fields 和 methods 能分别列出类的字段和方法，而 lock 则能使一个字段只读：

```
point_generator$fields()

##         x         y
## "numeric" "numeric"

point_generator$methods()

## [1] "add"                "callSuper"     "copy"
## [4] "distanceFromOrigin" "export"        "field"
## [7] "getClass"           "getRefClass"   "import"
## [10] "initFields"        "initialize"    "show"
## [13] "trace"             "untrace"       "usingMethods"
```

还有一些其他的方法可以从生成器对象或实例对象中调用。例如 show 能打印对象，trace 和 untrace 可让你在一个方法上使用 trace 函数，export 能把对象转换为另一个种类型，而 copy 则能生成拷贝。

参考类支持继承，其子类可扩展其功能。例如，我们可以创建一个新的、包含原来二维点类的三维点类，它包括了一个额外的 z 坐标。

类可以使用 contain 参数继承其他类的字段和方法：

```
three_d_point_generator <- setRefClass(
  "three_d_point",
  fields = list(
    z = "numeric"
  ),
  contains = "point",          # 这一行让我们继承
  methods = list(
    initialize = function(x, y, z)
    {
      "Assign x and y upon object creation."
      x <<- x
      y <<- y
      z <<- z
    }
  )
)
a_three_d_point <- three_d_point_generator$new(3, 4, 5)
```

此时，distanceFromOrigin 函数是错误的，因为它没有把 z 维度考虑在内：

```
a_three_d_point$distanceFromOrigin()  #错了！
```

```
## [1] 5
```

我们需要重写它，使它在新类中赋以新的意义。这可通过把相同的名称添加到类生成器中完成：

```
three_d_point_generator <- setRefClass(
  "three_d_point",
  fields = list(
    z = "numeric"
  ),
  contains = "point",
  methods = list(
    initialize = function(x, y, z)
    {
       "Assign x and y upon object creation."
       x <<- x
       y <<- y
       z <<- z
    },
    distanceFromOrigin = function()
```

```
      {
        "Euclidean distance from the origin"
        sqrt(x ^ 2 + y ^ 2 + z ^ 2)
      }
    )
  )
```

为了使用新的定义，我们需要重新创建我们的点：

```
a_three_d_point <- three_d_point_generator$new(3, 4, 5)
a_three_d_point$distanceFromOrigin()
```

```
## [1] 7.071
```

有时候，我们想要使用父类（又名*超类*）的方法。`callSuper` 方法正是做这个用的，因此可（低效地）重写 3D `distanceFromOrigin` 代码为：

```
distanceFromOrigin = function()
{
  "Euclidean distance from the origin"
  two_d_distance <- callSuper()
  sqrt(two_d_distance ^ 2 + z ^ 2)
}
```

面向对象编程是一个很大的话题，即使仅限于引用类，它完全可独立成书。John Chambers（S 语言创建者、R 核心成员以及参考类代码的作者）目前正在写一本关于 R 的面向对象编程的书。在落实之前，`?ReferenceClasses` 的帮助页面是目前最权威的参考类的参考。

16.8 小结

- R 有三种问题反馈层次：消息、警告和错误。
- 把代码包括在 `try` 或 `tryCatch` 中能让你更好地控制错误处理。
- `debug` 及其相关函数能帮助你调试函数。
- `RUnit` 和 `testthat` 包能让你进行单元测试。
- R 程序包含了被称为调用和表达式的语言对象。
- 你可以把字符串转换成语言对象，反之亦然。
- R 中有六个不同的面向对象的编程系统，但对于新项目只需要 S3 和引用类。

16.9 知识测试：问题

- 问题 16-1
  如果你的代码生成了超过 10 个的警告，你会使用什么函数来查看它们？

- 问题 16-2
  如果错误被抛出，调用 `try` 的返回值类型是什么？

- 问题 16-3
  要在 RUnit 测试中测试一个错误是否被正确抛出，你可以调用 `checkException` 函数。在 `testthat` 中与之相等的函数是什么？

- 问题 16-4
  你需要使用哪两个函数在命令行中模仿代码的执行？

- 问题 16-5
  怎么做才能让你的 `print` 函数对不同类型的输入做不同的事情？

16.10　知识测试：练习

- 练习 16-1
  调和平均值被定义为数据的倒数的平均值的倒数，即 `1 / mean(1 / x)`，其中 `x` 为正数。编写一个调和平均函数，当输入不是数字或含有非正数值，它能提供适当的反馈。[10]

- 练习 16-2
  使用 `RUnit` 或 `testthat` 为你的调和平均值函数编写一些测试。你应该检查：1、2 和 4 这几个数的调和平均数应该等于 `12/7`；没有输入时应该抛出一个错误，传递缺失值应该表现正常；对于非数字和非正数输入也应该与期望相同。继续测试，直到所有的测试都通过！ [15]

- 练习 16-3
  修改你的调和平均值函数，使其返回一个 `harmoinc` 的类。现在，为此类编写一个 S3 的 `print` 方法，使它能显示“The harmonic mean is y”的消息，其中 `y` 是调和平均值。[10]

第 17 章

制作程序包

R 的成功在于它的社区。R 的核心团队工作很出色，但重要的是，大多数的 R 代码是由用户们编写的。在本章中，你将学习如何创建自己的软件包以及如何与你的同事、朋友和更广阔的外部世界分享你的代码。即使你是一个不喜欢分享、独自工作的隐士，包也是一个能帮助你组织代码的好工具。

17.1 本章目标

阅读本章后，你会了解以下内容：

- 如何创建一个包；
- 如何为包中的函数和数据集撰写文档；
- 如何把包发布到 CRAN 上去。

17.2 为什么要创建软件包

要分享 R 程序，使其能被他人重用（或仅你自己），最自然的方式是将它打包。根据我的经验，很多 R 的用户没有尽早学习如何创建自己的包，觉得它高不可攀。而实际上，它非常简单，只要你按照预先定好的规则来做即可。这些规则都记录在 R 随附的名为 Writing R Extensions 的手册上。如果出了错，总能在那些文档中找到答案。

17.3 先决条件

构建包需要一堆运行在 Linux 和其他 Unix 系列平台上的标准工具，不过没有 Windows 平

台。所有的工具都被收集在一个链接中，可在 http://cran.r-project.org/bin/windows/Rtools（或离你最近的 CRAN 镜像的 bin/windows/Rtools 目录中）找到它们 。为了使安装更加容易，你可以使用 `installr` 包中 `install.Rtools`。

当安装东西时，你可能会需要 `devtools` 和 `roxygen2` 包：

```
install.packages(c("devtools", "roxygen2"))
```

17.4 包目录结构

创建一个包基本上就是要把正确的文件放在正确的地方。在你的包目录下，有两个文件是必须的：

(1) DESCRIPTION 包含了有关包的版本、作者及其用途。
(2) NAMESPACE 说明哪些函数将被输出（给用户）。

其他三个文件是可选的：

(1) LICENSE（或 LICENCE，这取决于你站在哪个阵营）包含了包的许可证。
(2) NEWS 即新闻信息，在此你可以记录所有你所做出的、令人激动的更新信息。
(3) INDEX 包含了包中所有有趣的东西的名称和描述。

如果你对要编写所有的五个管理文件感到焦虑，请先深呼吸一下。其实 NAMESPACE 和 INDEX 完全是自动生成的，而 DESCRIPTION 是部分自动生成的。而且，如果你只使用几种常用且标准的许可，那么就不需要许可证文件了[1]。

在顶层必须包含两个目录：

(1) R 目录包含你的 R 程序。
(2) man 目录包含了帮助文件。

还有一些可选的目录：

(1) src 包含 C、C++ 或 Fortran 的源代码。
(2) demo 包含可由 R 的 `demo` 函数运行的演示代码。
(3) vignettes 包含较长的文档，它将解释如何使用该程序包，这可以通过 `browseVignettes` 来查到。
(4) doc 中包含其他格式的帮助文件。
(5) data 包含数据文件。
(6) inst 包含其他的任何东西。

注 1：NEWS 非常令人讨厌，你将总是忘记更新它。

第一个可选的目录已经超出了本节关于快速创建包的范围。在三种编译语言中，C++ 最容易和 R 一起使用，这要归功于 `Rcpp` 包（参见 Dir Eddelbuettel 的 *Seanless R and C++ Integration with RCpp*）。创建 Vignettes 并不难，特别是如果你使用了 `knitr` 包（请阅读 Yihui Xie 的 *Dynamic Documents with R and Knitr*）。

data 文件可以通过 `data` 函数得到的文件（正如我们在 12.2 节所看到的）。它们的首选格式是 `.RData` 文件，即调用 `save` 的结果，虽然其他格式也是可以的。

虽然 inst 是一个可以包含任何东西的文件夹，但它也可以包含一些标准的内容：

(1) `inst/test` 含有你的 `RUnit` 或 `testthat` 测试。
(2) inst/python、inst/matlab 和 inst/someotherscriptinglanguage 包含其他脚本语言代码。（三种已经支持的编译语言则位于 src 目录下，如前所述）。
(3) 你可以在 CITATION 文件中描述希望包如何被引用，虽然这些信息通常从描述文件中自动生成。

17.5 你的第一个包

好了，理论谈到这里为止，让我们来试着生成一个包吧。首先我们需要包括一些东西——上一章的 `hypotenuse` 函数就很好。为了演示如何把数据包含到包里面，我们使用了一些毕达哥拉斯三元组：

```
hypotenuse <- function(x, y)
{
  sqrt(x ^ 2 + y ^ 2)
}
pythagorean_triples <- data.frame(
  x = c(3, 5, 8, 7, 9, 11, 12, 13, 15, 16, 17, 19),
  y = c(4, 12, 15, 24, 40, 60, 35, 84, 112, 63, 144, 180),
  z = c(5, 13, 17, 25, 41, 61, 37, 85, 113, 65, 145, 181)
)
```

那么现在我们应该创建一些目录，以记录东西放在哪儿，对吗？事实上，不用这么复杂。`package.skeleton` 函数能创建（几乎）所有我们需要的东西。它的第一个参数是包的名字（“pythagorus”就挺好），第二个参数是一个字符向量，里面包含你要添加的变量名：

```
package.skeleton(
  "pythagorus",
  c("hypotenuse", "pythagorean_triples")
)
```

运行 `package.skeleton` 将创建 R、man 和 data 目录，DESCRIPTION 和 NAMESPACE 文件，以及一个名为 Read-and-delete-me 的文件，它包含了进一步的说明。其输出如图 17-1 所示。

```
> package.skeleton(
+   "pythagorus",
+   c("hypotenuse", "pythagorean_triples")
+ )
Creating directories ...
Creating DESCRIPTION ...
Creating NAMESPACE ...
Creating Read-and-delete-me ...
Saving functions and data ...
Making help files ...
Done.
Further steps are described in './pythagorus/Read-and-delete-me'.
> |
```

图 17-1：使用 package.skeleton 创建 pythagorus 包

DESCRIPTION 文件有一个严格的结构 *Name: value* 对。我们只需要更新 Title、Author、Maintainer、Description 和 License 字段为合适的内容。基本的文件格式，由 package.skeleton 创建的，如图 17-2 所示：

```
DESCRIPTION
Package: pythagorus
Type: Package
Title: What the package does (short line)
Version: 1.0
Date: 2013-05-13
Author: Who wrote it
Maintainer: Who to complain to <yourfault@somewhere.net>
Description: More about what it does (maybe more than one line)
License: What license is it under?

```

图 17-2：由 package.skeleton 创建的 DESCRIPTION 文件

License 字段必须是一个文件（本例中必须包括一个 LICENSCE 或 LICENSE）、“Unlimited”（没有限制）或者以下标准许可之一：GPL - 2、GPL-3、LGPL-2、LGPL-2.1、LGPL-3、AGPL-3，Artistic-2.0、BSD_2_clause、BSD_3_clause 或 MIT。

NAMESPACE 文件包含了文本 exportPattern("^[[:alpha:]]+")。它的意思是“任何命名以字母开头的变量都可以给用户用”。现在流行的最佳实践是为每个你想要提供的变量写一个 export 语句，而不是指定一个模式。

我们将在下一节看到 NAMESPACE 如何被自动创建，现在只需要把其中的文本替换成 export(hypotenuse)。

man 目录包含了一些自动生成的 .Rd 文件：一个用于每个函数，一个用于每个数据集，还有一个文件命名为 pythagorus-package.Rd。一旦包被建立起来，这些包含了 LaTeX 标记的 .Rd 文件将被用于创建帮助页面。pythagorus-package.Rd 文件包含了整个包的帮助页面，你可以在那里介绍包的其余部分的内容。如图 17-3 所示，这是一自动生成的 .Rd 文件的例子。

```
\name{hypotenuse}
\alias{hypotenuse}
%- Also NEED an '\alias' for EACH other topic documented here.
\title{
%%  ~~function to do ... ~~
}
\description{
%%  ~~ A concise (1-5 lines) description of what the function does. ~~
}
\usage{
hypotenuse(x, y)
}
%- maybe also 'usage' for other objects documented here.
\arguments{
  \item{x}{
%%     ~~Describe \code{x} here~~
}
```

图 17-3：为 hypotenuse 函数自动生成的帮助页面的源文件

虽然基本格式已经被自动创建出来，但是 R 不知道函数到底是什么，你需要手动填写一些细节。对此有两种方法，一种比较难，而另一种简单些，下节将讨论后者。

17.6　为包撰写文档

把函数的帮助页面放在不同目录的最大的问题是很容易使内容不同步。这方面的一个典型例子是当你要添加、删除或重命名函数参数时，R 不能自动更改相应的帮助文件来匹配函数的更改，你必须同时修改两个文件才能确保它们之间能一直保持更新同步。

`roxygen2` 包通过让你在 R 的代码注释中直接使用帮助文本解决了这个问题。它还能通过使用一个简单的标记语言，大大减少了对 LaTeX 知识的依赖。`roxygen2` 继承自 `Doxygen`，它是一个能生成类似 C++、C、Java、Fortran、Python 和其他语言文档的工具。它的语法非常值得学习，使你能为其他许多语言生成文档。

`roxygen2` 标记的每一行以 `#'` 开头。有些部分，如标题和描述，是通过它们在注释段的开始位置来标记。其他部分则以关键字标注。例如，描述返回值的部分使用 `@return` 开头来标注。一个完整的帮助文本段如下所示：

```
#' Help page title
#'
#' A couple of lines of description about the function(s).
#' If you want to include code, use \code{my_code()}.
#' @param x Description of the first argument.
#' @param y Description of the second argument.
#' @return Description of the return value from a function.
#' If it returns a list, use
#' \itemize{
```

```
#'    \item{item1}{A description of item1.}
#'    \item{item2}{A description of item2.}
#' }
#' @note Describe how the algorithm works, or if the function has
#' any quirks here.
#' @author Your name here!
#' @references Journal papers, algorithms, or other inspiration here.
#' You can include web links like this
#' \url{http://www.thewebsiteyouarelinkingto.com}
#' @seealso Link to functions in the same package with
#' \code{\link{a_function_or_dataset}}
#' and functions in other packages with
#' \code{\link[another_package]{a_function_or_dataset}}
#' @examples
#' #R code run by the example function
#' \dontrun{
#' #R code that isn't run by example or when the package is built
#' }
#' @keywords misc
#' @export
f <- function(x, y)
{
  #Function content goes here, as usual
}
```

在上例中，有几件事情需要特别注意。

参数需要使用 `@parm` 关键字标注。（术语“parm”在整个 Doxygen 系统中都是标准的，所以如果想在 R 中把它改为“arg”会带来更多的混乱，还不如删除它。）`@parm` 参数后面跟着一个空格、参数名、另一个空格，以及参数的描述。

范例中的任何东西都必须是合法的 R 程序，因为当你构建软件包时它会自动运行。如果你想添加注释，请使用一个额外的哈希符 `#`（在现有的用于 `roxygen2` 的 `#'` 号顶部）来创建。如果你想补充一些失败情景下（例如演示创建文件时发生错误）的范例，可以把它们包装在一个 `\dontrun{}` 块中。

帮助文件可包含关键字，但不是所有关键字都可以。请安装 `R.oo` 包来查看所有可能值的列表，并运行此代码片段：

```
library(R.oo)
Rdoc$getKeywords()
```

（或者也可以打开 `R.home ("doc")` 返回目录中的 `KEYWORD` 的文件。）

添加 `@export` 关键字将列出 `NAMESPACE` 文件中的函数，而这意味着用户应该能够从此包中调用该函数，而不仅仅是一个内部的辅助函数。

整个包的描述文档位于一个名为 `packagename-package.R` 的文件中。它类似于函数的文档，但可能更容易撰写，因为它的内容更加少：

```
#' Help page title. Probably the package name and tagline.
#'
#' A description of what the package does, why you might want to use it,
#' which functions to look at first, and anything else that the user
#' really, absolutely, must look at because you've created it and it is
#' astonishing.
#'
#' @author You again!
#' @docType package
#' @name packagename
#' @aliases packagename packagename-package
#' @keywords package
NULL
```

函数文档中有两处位非常重要，一个是 `@docType package` 行，它告诉 roxygen2 这是整个包的文档；另一个是最后的 `NULL` 值，它的出现是由于技术的原因——如果你忽略它就会产生错误。

为数据集撰写文档几乎与为整个包撰写文档一样。对此文档的存放位置并没有特定的标准，你可以把它追加到包文档后面，又或者把它放到一个单独创建的 packagename-data.R 文件中：

```
#' Help page title
#'
#' Explain the contents of each column here in the description.
#' \itemize{
#'    \item{column1}{Description of column1.}
#'    \item{column2}{Description of column2.}
#' }
#'
#' @references Where you found the data.
#' @docType data
#' @keywords datasets
#' @name datasetname
#' @usage data(datasetname)
#' @format A data frame with m rows of n variables
NULL
```

和包一样，对于数据集来说有两处很重要：告诉 roxygen2 这是函数文档的 `@docType data` 行，以及最后的 `NULL` 值。

在你为每个函数、数据集和整个包撰写了文档之后，请调用 `roxygenize` 函数生成帮助文件以及更新 NAMESPACE 和 DESCRIPTION 文件（对于某些更喜欢英式拼写法的人来说，可以使用 `roxygenize` 来替代 `roxygenise`）：

```
roxygenize("path/to/root/of/package")
```

17.7 检查和构建包

现在，你已经创建了所有必需的目录，添加了 R 代码和数据集，并为它们撰写了文档。你马上就可以开始准备建立你的包了——最后一个任务是检查一切是否正常 [2]。

R 有一个内置的检查工具 `R CMD check`，你可以从操作系统的命令行中使用它。它检查非常彻底，这也是那些你从 CRAN 下载的大部分软件包能实际工作的主要原因。当然，使用 DOS 或 bash 命令行就像是还工作在二十世纪一样——更好的替代方法是使用 `devtools` 包中的 `check` 函数，它的输出如图 17-4 所示：

```
library(devtools)
check("path/to/root/of/package")
```

```
> check("pythagorus")
Updating pythagorus documentation
Loading pythagorus
Writing pythagorean_triples.Rd
"C:/PROGRA~1/R/R-devel/bin/x64/R" --vanilla CMD build  \
  "d:\workspace\pythagorus" --no-manual --no-resave-data

* checking for file 'd:\workspace\pythagorus/DESCRIPTION' ... OK
* preparing 'pythagorus':
* checking DESCRIPTION meta-information ... OK
* checking for LF line-endings in source and make files
* checking for empty or unneeded directories
Removed empty directory 'pythagorus/inst'
* looking to see if a 'data/datalist' file should be added
* building 'pythagorus_1.0.tar.gz'

"C:/PROGRA~1/R/R-devel/bin/x64/R" --vanilla CMD check  \
```

图 17-4：检查包时的输出

这会输出好几个页面，并对某些东西发出警告，例如文件与相应的函数不匹配、命名并非跨平台的、范例无法正确运行以及它认为你的编码风格过时了。（好吧，最后一个是编造的，但是那里的确有很多的检查。）

仔细阅读其输出并修正错误和警告，然后再来一次。一旦你确信你的包文件已经没有错误，那么你终于可以构建它了！除了 `check` 以外，R 还有一个内置的命令行版本的 `build`，但是使用 `devtools` 包里的函数会容易得多。你可以选择构建成源代码（这样会具有跨平台的可移植性，是 Linux 的标准格式）或二进制（特定于你的当前操作系统）：

```
build("path/to/root/of/package")
```

注 2：在此先泼一盆冷水，答案通常是："等等，你还有些事情没完成。"

成功了，你现在拥有了自己的包！不过，如果其他人也可以使用你的包，那岂不是更好？要把你的包发布到 CRAN 上，可以使用 devtools 的 release 函数：

```
release("path/to/root/of/package")
```

它会问你很多问题，以确保在发布包之前，你真的万分肯定已完成了所需要的事情。

> 当你把包上传到 CRAN 时，R 核心成员将检查你的包是否在构建时有错误。他们的时间很宝贵，所以在把包上传到 CRAN 之前，请务必确保你已运行了 check 函数并且修复了所有错误和警告。

17.8 包的维护

函数对用户来说就像是一个黑盒子。他们会给它传递一些参数然后函数也会返回一些值，用户不需要关心函数内部发生了什么（至少理论上如此）。这意味着，一个函数的签名（即函数的名称及参数的顺序）不应该没有警告用户就直接更改。R 提供了一些函数能帮助通知用户签名已改变。

如果你正计划增加一个新的功能但还没有机会实现它，或想提前通知你的用户此功能将至，可以使用 .NotYetUsed 函数。如果用户试图过早地使用它，这将抛出一个错误或警告，理由是它还不能使用。在下例中，我们将为 hypotenuse 函数添加一个二维的 p 范数。在添加新的功能之前，我们只需更改其签名，并且在用户尝试使用 p 参数时抛出一个错误：

```
hypotenuse <- function(x, y, p = 2)
{
  if(!missing(p))
  {
    .NotYetUsed("p")
  }
  sqrt(x ^ 2 + y ^ 2)
}
hypotenuse(5, 12)     # 行为与之前的一样

## [1] 13

hypotenuse(5, 12, 1)

## Error: argument 'p' is not used (yet)
```

一旦我们添加了新的功能，可以删除 .NotYetUsed：

```
hypotenuse <- function(x, y, p = 2)
{
  (x ^ p + y ^ p) ^ (1 / p)
}
```

如果你想添加一个全新的功能（而不仅仅是一个参数），对应的函数是 .NotYetImplemented。当你第一次创建包或添加一大块功能时这非常适用。编写单个函数会非常耗时，所以当写完一些代码，你可能已忘记其他本来要添加的函数。因此，有时最好是先完成上层的设计，然后再逐步实现具体的底层细节。只需简单地为每个函数先创建一个占位符，在函数体中先使用 .NotYetImplemented 代替。下例中的函数将会在以后某一天才开始计算三角数，不过它现在只会抛出一个错误：

```
triangular <- function(n)
{
  .NotYetImplemented()
}
triangular()

## Error: 'triangular' is not implemented yet
```

如果你想删除一个函数，最礼貌的做法是是分阶段完成。第一步是在函数内添加一个 .Deprecated 的调用，其参数为它所要替代的函数的名称。该函数的其余部分应保持不变，从而能够保留现有的行为：

```
hypotenuse <- function(x, y, p = 2)
{
  .Deprecated("p_norm")
  (x ^ p + y ^ p) ^ (1 / p)
}
hypotenuse(5, 12)

## Warning: 'hypotenuse' is deprecated. Use 'p_norm' instead. See
## help("Deprecated")
## [1] 13
```

经过一段适当长的时间——足以让你的用户注意到相关函数过时的消息，你就可以改变函数的内容，让它调用 .Defunct，这将抛出一个错误：

```
hypotenuse <- function(x, y, p = 2)
{
  .Defunct("p_norm")
}
hypotenuse(5, 12)

## Error: 'hypotenuse' is defunct. Use 'p_norm' instead. See help("Defunct")
```

17.9 小结

- 包的制作主要涉及如何按特定的结构来组织文件。
- package.skeleton 会为你创建这种结构中大部分框架。
- roxygen2 包使包的文档制作非常容易。
- devtools 包使包的检查和构建非常容易。

- `NotYetImplemented`、`Deprecated` 和 `Defunct` 能帮助你维护包。

17.10 知识测试：问题

- 问题 17-1
  R 包顶层目录中的五个文件有哪几个是必须的？这些文件是 DESCRIPTION、INDEX、LICENSE、NAMESPACE 和 NEWS。

- 问题 17-2
  R 包中的八个目录中哪些是必须的？这些目录是 data、demo、doc、inst、man、R、src 和 vignettes。

- 问题 17-3
  为什么需要在你的软件包中包含一个 CITATION 文件？

- 问题 17-4
  你可以调用哪个函数从 `roxygen2` 标记中生成帮助文件？

- 问题 17-5
  如何才能比较礼貌地从包中删除一个函数？

17.11 知识测试：练习

- 练习 17-1
  写一个函数 `sum_of_squares`，它能计算的前 `n` 个平方数之和。（提示：公式为 `n *（n + 1 ）* ( 2 * n + 1 ) / 6`）[5]

  创建一个有两列的数据框 `squares_data`。其 `x` 列应包含数字 1 到 10。`y` 列为 `x` 列中对应行的平方和—这也就是 `sum_of_squares(1:10)` 的结果。[5]

  使用 `package.skeleton` 创建 `square` 包，它包含了 `sum_of_squares` 函数和 `squares_data` 数据框。[5]

- 练习 17-2
  为 `sum_of_squares` 函数和 `squares_data` 数据框撰写文档，并使用 `roxygen2` 标记来为整个 `squares` 包撰写文档并生成帮助文件。

- 练习 17-3
  使用 `devtools` 包检查并构建 `squares` 包。

第三部分

附录

附录A

变量的属性

R 有四种不同的函数能告诉你变量的类型是什么，即 `class`、`typeof`、`mode` 和 `storage.mode`。对于某些变量类型，这些函数给出的答案都相同，但对于其他的函数则有点不同，这让事情变得有点复杂。

对于绝大部分你所编写的代码，你只需关心其 `class` 类型。`class` 唯一不能帮你的是在检查矩阵或数组时，变量是否包含数字或字符（或其他类型）。在这种情况下，你可以使用其他三种类型的函数之一，或只调用 `is.*` 函数（例如 `is.numeric`）中的一个。

表 A-1 显示了 `class`、`typeof`、`mode` 和 `storage.mode` 对各种变量类型返回的值。

表A-1：不同类、类型、模式和存储模式之间的比较

	class	typeof	mode	storage.mode
Logical	logical	logical	logical	logical
Integer	integer	integer	numeric	integer
Floating Point	numeric	double	numeric	double
Complex	complex	complex	complex	complex
String	character	character	character	character
Raw byte	raw	raw	raw	raw
Categorical	factor	integer	numeric	integer
Null	NULL	NULL	NULL	NULL
Logical Matrix	matrix	logical	logical	logical
Numeric Matrix	matrix	double	numeric	double
Character Matrix	matrix	character	character	character

（续）

	class	typeof	mode	storage.mode
Logical Array	array	logical	logical	logical
Numeric Array	array	double	numeric	double
Character Array	array	character	character	character
List	list	list	list	list
Data Frame	data.frame	list	list	list
Function	function	closure	function	function
Environment	environment	environment	environment	environment
Expression	expression	expression	expression	expression
Call	call	language	call	language
Formula	formula	language	call	language

在 R 中，向量是具有长度但没有维度的（即 `dim` 会返回 `NULL`），以及除了名字以外没有属性的变量类型。向量类型包括了数值、逻辑和字符类型，而且还包括列表和表达式。无属性的规则意味着因子不是向量。

列表是向量。因子不是向量。

与向量相关的是原子（atomic）类型。*原子*表示此类型不可以把相同类型的其他实例包含在内。与原子相反的类型是*递归*（recursive）：列表就是典型的例子，因为它可以把其他列表包含在内。一个对象永远只能是原子或递归的，不可能兼顾。

矩阵和数组都是原子的。

有些对象被称为*语言对象* (language objects)。这些变量类型可被解析为 R 程序运行。

表 A-2 显示了 `is.vector`，`is.atomic`，`is.recursive` 和 `is.language` 对各种变量类型的返回值。

表A-2：矢量、原子、递归或语言对象变量类型的比较

	is.vector	is.atomic	is.recursive	is.language
Logical	TRUE	TRUE	FALSE	FALSE
Integer	TRUE	TRUE	FALSE	FALSE
Floating Point	TRUE	TRUE	FALSE	FALSE
Complex	TRUE	TRUE	FALSE	FALSE

（续）

	is.vector	is.atomic	is.recursive	is.language
String	TRUE	TRUE	FALSE	FALSE
Raw Byte	TRUE	TRUE	FALSE	FALSE
Categorical	FALSE	TRUE	FALSE	FALSE
Null	FALSE	TRUE	FALSE	FALSE
Logical Matrix	FALSE	TRUE	FALSE	FALSE
Numeric Matrix	FALSE	TRUE	FALSE	FALSE
Character Matrix	FALSE	TRUE	FALSE	FALSE
Logical Array	FALSE	TRUE	FALSE	FALSE
Numeric Array	FALSE	TRUE	FALSE	FALSE
Character Array	FALSE	TRUE	FALSE	FALSE
List	TRUE	FALSE	TRUE	FALSE
Data Frame	FALSE	FALSE	TRUE	FALSE
Function	FALSE	FALSE	TRUE	FALSE
Environment	FALSE	FALSE	TRUE	FALSE
Expression	TRUE	FALSE	TRUE	TRUE
Call	FALSE	FALSE	TRUE	TRUE
Formula	FALSE	FALSE	TRUE	TRUE

附录B

R中其他可做的事情

还有很多 R 可以做的事情本书没有加以介绍，有的是因为它们的需求比较小众，有的是因为过于高级，还有的是因为有关那些主题已有了非常好的介绍书籍。本附录旨在告诉你如何找到这些资源。有好几个这里涉及的主题都能在 CRAN Task View（http://cran.r-project.org/web/views/）里找到，非常值得你仔细浏览。

你可以在 R 代码中使用 GUI 来帮助那些不太懂技术的用户。例如，`gWidgets` 框架允许从高层访问几个图形工具包，其中包括 `tcl/tk`、`qt`、`GTK` 和 `HTML`（`tcltk/tcltk2`、`qtbase`、`RGtk2` 和几个 HTML 生成器的包也可以进行底层访问）。参阅 Michael Lawrence 和 John Verzani 的 *Programming Graphical User Interface in R* 来获取更多的信息吧。基于 Java 的 `deducer` 包提供了一个替代，而 RStudio 的 `shiny` 包则使用 R 来编写网络应用程序变得非常简单。

你可以通过 `.Call` 函数调用编译语言的代码（如 C、C++ 和 FORTRAN）。到目前为止，使用这些语言的最简单的方法是编写 C++ 的代码并且与 `Rcpp` 包一起使用。关于此主题的更多信息，请参阅 Dirk Eddelbuetted 的 *Seamless R and C++ Integration with Rcpp*。

R 是单线程的，不过也有好几种方法能够让它利用多核或多台机器来并行运行你的代码。通过使用 R 中附带的 `parallel` 包，在符合 POSIX 协议的操作系统（这些操作系统都不属于 Windows）下，你可以直接用 `mclapply` 取代 `lapply`，使之在你机器上的所有内核上执行循环。通过使用 `snow` 和 `parallel` 包还可得到基于套接字的多核集群功能（例如 MPI、SOCK 或其他）。你也可以使用 RHIPE（http://www.datadr.org/）和 `rmr` 包连接到一个 Hadoop 集群，或使用 `segue` 连接到 Amazon Web Services 下的 hadoop，但注意这并不适用于 Windows 平台。更多细节请阅读 Q.Ethan McCallum 和 Stephen Weston 著写的 *Parallel R*。

Revolution R 的企业版具有内置的一些并行化功能。

ff 和 bigmemory 包允许你把 R 的变量存储到文件中（如 SAS 一样），这就避免了内存的限制。与此相关的 data.table 包提供了增强的数据框变量类型，它具有更快的索引、分配和合并功能。

R 还有几十个软件包可用于空间统计：sp 提供了存储空间数据对象的标准方式，maps、maptools 和 mapproj 提供用于读写地图的辅助函数，spatstat 提供了空间统计的函数（你猜对了），还有 OpenStreetMap 能从 http://openstreetmap.com 检索光栅图像，等等。

最后，R 有用于组合代码以及使用常规文本输出到报告中（有时也被称为文学编程或重复性的研究）的工具，即 Sweave。它已经由 knitr 包改善和扩展，它将允许你使用各种标记语言来创建报告。事实上，本书就是用 knitr 创建 AsciiDoc 标记，从而用于创建 PDF、HTML 和 ebook 的文档。Yihui Xie 的 *Dynamic Documents with R and knitr* 解释了如何使用它。

R 的生态系统很大，并随着时间的推移变得越来越大。这些可能只是你想探索的一小部分东西而已；请阅读一下参考书目，看是否有令你感兴趣内容。愉悦地享受探索之旅吧！

附录C

问题答案

- 问题 1-1

  R 是 S 编程语言的开源版本。

- 问题 1-2

  选项包括：命令式、面向对象和函数式。

- 问题 1-3

  `8:27`

- 问题 1-4

  `help.search`（这与 `??` 的效果一样）

- 问题 1-5

  `RSiteSearch`

- 问题 2-1

  `%/%`

- 问题 2-2

  `all.equal (x, pi)`，或者更好的答案是：`isTrue(all.equal(x, pi))`

- 问题 2-3

  至少下面的两个：

  (1) `<-`

(2) +=

(3) +<<-

(4) assign

- 问题 2-4

  只是 Inf 和 -Inf

- 问题 2-5

  0、Inf 和 -Inf

- 问题 3-1

  numeric、integer 和 complex

- 问题 3-2

  nlevels

- 问题 3-3

  as.numeric ("6.283185")

- 问题 3-4

  summary、head、str、unclass、attributes 或 View 中的任意三个。如果你还找出与 head 作用相反的——能打印出最后几行的 tail 函数则能加分。

- 问题 3-5

  rm(list = ls())

- 问题 4-1

  seq.int(0,1,0.25)

- 问题 4-2

  要么在创建矢量时使用 name = value 对，要么在之后使用 names 函数。

- 问题 4-3

  正整数的位置检索，负整数的位置检索，逻辑值或元素的名称。

- 问题 4-4

  3 * 4 * 5 = 60

- 问题 4-5

  %*%

- 问题 5-1

  长度为 3。因为内部列表只算是一个元素，NULL 元素也是一样。

- 问题 5-2

  当向函数传递参数时、调用 `formals` 时，或者在全局环境变量 `.Options` 里。

- 问题 5-3

  你可以在单方括号中结合矩阵式索引和正整数 / 负整数 / 逻辑值 / 字符来使用。你也可以在单或双括号中结合列表样式索引中和单个索引值使用，或者是使用美元符号（`$`）运算符。最后，你还可以使用 `subset` 函数。

- 问题 5-4

  通过把 `check.names = FALSE` 传递给 `data.frame`。

- 问题 5-5

  可以使用 `rbind` 添加到垂直方向上，使用 `cbind` 添加到水平方向。

- 问题 6-1

  用户工作区。

- 问题 6-2

  `list2env` 是最好的方案，不过 `as.environment` 也适用。

- 问题 6-3

  只需要输入其名称。

- 问题 6-4

  `formals`、`args` 和 `formalArgs`。

- 问题 6-5

  `do.call` 能以列表的形式把参数传递给函数。

- 问题 7-1

  主要就是 `format`、`formatC`、`sprintf` 和 `prettyNum`。

- 问题 7-2

  使用 `alarm` 或者把 `\a` 字符打印到控制台。

- 问题 7-3

  `factor` 或 `ordered`。

- 问题 7-4

  该值将被算作缺失值（`NA`）。

- 问题 7-5

  使用 `cut` 来将其分组。

- 问题 8-1

  如果你传入一个 `NA` 值，`if` 会抛出一个错误。

- 问题 8-2

  `Ifelse` 将在条件为 `NA` 处返回 `NA`。

- 问题 8-3

  `switch` 会基于字符或整型参数有条件地执行代码。

- 问题 8-4

  在你的循环代码中插入关键字 `break`。

- 问题 8-5

  在你的循环代码中插入关键字 `next`。

- 问题 9-1

  在各章中讨论过 `lapply`、`vapply`、`sapply`、`apply`、`mapply` 和 `tapply`，亦简要提及 `eapply` 和 `rapply`。试试 `apropos("apply")` 来查看它们所有的列表。

- 问题 9-2

  所有这三个函数都接受一个列表，并把函数依次应用于每个元素上。它们所不同的是返回值。`lapply` 总是返回一个列表，`vapply` 总是返回一个由模板指定的向量或数组，`sapply` 的返回值也一样。

- 问题 9-3

  `rapply` 是递归的，它非常适合访问诸如树这样的深层嵌套对象。

- 问题 9-4

  这是一个经典的拆分－应用－合并问题。使用 `tapply`（或 `plyr` 包中的某些函数）吧。

- 问题 9-5

  类似 `**ply` 的名称中，第一个星号表示第一个输入参数的类型，第二个星号表示返回值的类型。

- 问题 10-1

  CRAN 是迄今为止最大的包库。其他的还有 Bioconductor、R-Forge 和 RForge.net 等。还可以在 GitHub、bitbucket 和 Google Code 上找到很多包。

- 问题 10-2

  这两个函数都能装载一个包，但失败时 `library` 会抛出一个错误，而 `require` 则返回一个逻辑值（让你自定义错误处理逻辑）。

- 问题 10-3
  一个包库不过就是你机器上的一个包含了 R 包的文件夹而已。

- 问题 10-4
  `.libPaths` 能返回库的列表。

- 问题 10-5
  R 不能像 IE 浏览器一样工作，但你可以使用 IE 浏览器的的 `internet2.dll` 库用于连接互联网。

- 问题 11-1
  必须使用 `POSIXct` 类。Date 不能存储时间信息，而 `POSIXlt` 日期则把它们的数据存储为列表，这使它不能放入一个数据框里面。

- 问题 11-2
  1970 年 1 月 1 日开始的午夜。

- 问题 11-3
  `"%B %Y"`

- 问题 11-4
  给它加上 3600 秒。例如：

  ```
  x <- Sys.time()
  x + 3600
  ##[1]" 2013-07-17 22:44:55 BST "
  ```

- 问题 11-5
  周期会比较长，因为 2016 是闰年。一年的持续时间正是 `60 * 60 * 24 * 365` 秒。对于闰年来说一年的周期是 366 天。

- 问题 12-1
  调用不带参数的 `data` 函数。

- 问题 12-2
  `read.csv` 假设小数点是句号（句点）以及用逗号作为分隔符，而 `read.csv2` 则假设小数点为逗号且以分号为分隔符。`read.csv` 用于使用句号作为小数点来创建数据的区域中（例如大多数的英语语言区域）。`read.csv2` 则用于使用逗号作为小数点来创建数据的区域中（例如大多数的欧洲语言区域）。如果你对此不太确定，只需在文本编辑器中打开你的数据文件。

- 问题 12-3

  xlsx 包中的 read.xlsx2 函数是首选，不过在同一个包中还有 read.xlsx 可用。也可以使用在其他几个包中的不同函数。

- 问题 12-4

  你可以简单地把网址传给 read.csv，或使用 download.file 得到一个本地副本。

- 问题 12-5

  目前均支持 SQLite、MySQL、PostgreSQL 和 Oracle 数据库。

- 问题 13-1

  使用 readLines 把文本读取为字符向量，然后调用 str_count 来计算此单词在每行出现的数量，最后用 sum 求总和。

- 问题 13-2

  with、within、transform 和 mutate 所有这些函数和标准的赋值函数一样，都允许操纵列和把列添加到数据框中。

- 问题 13-3

  是铸造（Casting），而不是冻结！

- 问题 13-4

  使用 order 或 arrange。

- 问题 13-5

  先定义一个函数，当你有一个正数—例如，当 is.positive <- function(x) x> 0 时返回 TRUE 值—然后调用 Find(is.positive, x)。

- 问题 14-1

  min 返回其所有输入中的单个最小值。pmin 接受若干长度相同的向量，并返回它们在每个位置上的最小值。

- 问题 14-2

  通过 pch("plot character") 参数。

- 问题 14-3

  使用形式为 y ~ x 的公式。

- 问题 14-4

  Aesthetic 指定一个你能看到变化的变量。大多数绘图会分别为 x 和 y 坐标取相应的 x 和 y aesthetic。你也可以指定颜色或形状的 aesthetic（例如，其中两个以上的变量会被同时看到）。

- 问题 14-5

  在本章中所提及的直方图、箱线图以及核密度图均可。还有其他一些古怪而艰深的绘图，例如没有提及的小提琴图（violin plot）、轴须图（rug plot）、豆荚图（bean plot）、茎叶图（stem-and-leaf plot）等。这些图当中，你每猜中一个，就可以获得 100 点极客分。

- 问题 15-1

  先设置指定的种子数（使用 `set.seed`）以生成一些随机数字，然后将种子值重置为原来的值。

- 问题 15-2

  PDF 的函数名字以 `d` 开头，后面跟着分布的名称。例如，PDF 格式的二项式分布函数是 `dbinom`。 CDF 函数以 `p` 开头，随后也是分布的名称，同样逆 CDF 函数也是以 `q` 开头随后跟着分布的名称。

- 问题 15-3

  冒号表示变量之间的交互。

- 问题 15-4

  `anova`、`AIC`、`BIC` 都是用于模型比较的常见函数。

- 问题 15-5

  R^2 值可以从 `summary(model)$r.squared` 中取得。

- 问题 16-1

  `warnings` 函数可显示之前的警告。

- 问题 16-2

  失败时，`try` 会返回一个 `try-error` 类的对象。

- 问题 16-3

  在 `testthat` 中相当于 `checkException` 的是 `expect_exception`。请立马说出来。

- 问题 16-4

  `quote` 把一个字符串转换成一个调用，然后 `eval` 会计算它。

- 问题 16-5

  使用 S3 系统的重载函数。`print.foo` 函数将会被 `foo` 类的对象调用。

- 问题 17-1

  DESCRIPTION 和 NAMESPACE 是必须的。

- 问题 17-2

  man 和 R 在所有软件包中都是必须的。如果你要包括 C、C + + 或 Fortran 的代码，那么 src 也是必要的。

- 问题 17-3

  CITATION 文件让你解释是谁编写以及是谁维护包的，如果该信息是太长或太复杂，可以放到 DESCRIPTION 文件中。

- 问题 17-4

  `roxygenise` 或 `roxygenize`

- 问题 17-5

  首先通过添加调用 `.Deprecated` 来警告用户此函数将过时。然后，使用调用 `Defunct` 完全替代函数体。

附录D

练习答案

- 练习 1-1

  如果你遇到困难，请咨询你的系统管理员，或者在 R-help 邮件列表上提问。

- 练习 1-2

  使用冒号操作符来创建一个向量，然后使用 sd 函数：

  ```
  sd(0:100)
  ```

- 练习 1-3

  输入 demo(plotmath) 并按下回车键，或者点击图形看看它能提供什么。

- 练习 2-1

  (1) 简单的除法就可以得到倒数，然后使用 atan 计算反正切（arc）：

  ```
  atan( 1 / 1:1000 )
  ```

  (2) 使用 <- 来给变量赋值：

  ```
  x <- : - 1:1000
  y <- atan(1/x)
  z <- 1 /tan(y)
  ```

- 练习 2-2

  对于比较两个应该包含相同数字的向量，通常 all.equal 就是你所需要的：

  ```
  x == z
  identical(x, z)
  ```

```
all.equal(x, z)
all.equal(x, z, tolerance = 0)
```

- 练习 2-3

  包含在以下三个向量的精确值可能不同：

  ```
  true_and_missing <- c(NA, TRUE, NA)
  false_and_missing <- c(FALSE, FALSE, NA)
  mixed <- c(TRUE, FALSE, NA)

  any(true_and_missing)
  any(false_and_missing)
  any(mixed)
  all(true_and_missing)
  all(false_and_missing)
  all(mixed)
  ```

- 练习 3-1

  ```
  class(Inf)
  class(NA)
  class(NaN)
  class("")
  ```

  把 `class` 代替为 `typeof`、`mode` 和 `storage.mode` 重复以上例子。

- 练习 3-2

  ```
  pets <- factor(sample(
    c("dog", "cat", "hamster", "goldfish"),
    1000,
    replace = TRUE
  ))
  head(pets)
  summary(pets)
  ```

  建议转换为因子，但这并非强制。

- 练习 3-3

  ```
  carrot <- 1
  potato <- 2
  swede <- 3
  ls(pattern = "a")
  ```

  你定义的蔬菜可能会有所不同。

- 练习 4-1

  创建序列有几种方法，其中包括冒号运算符。此解决方案使用 `seq_len` 和 `seq_along`：

```
n <- seq_len(20)
triangular <- n * (n + 1) / 2
names(triangular) <- letters[seq_along(n)]
triangular[c("a", "e", "i", "o")]
```

- 练习 4-2

  同样地，创建从 -11 序列为 0 再到 11 的序列有许多种不同的方法。abs 能计算一个数的绝对值：

```
diag(abs(seq.int(-11, 11)))
```

- 练习 4-3

  威尔金森（Wilkinson）矩阵有一个有趣的属性，它们的大部分特征值几乎相等。21x21 矩阵是最经常使用的：

```
identity_20_by_21 <- diag(rep.int(1, 20), 20, 21)
below_the_diagonal <- rbind(0, identity_20_by_21)
identity_21_by_20 <- diag(rep.int(1, 20), 21, 20)
above_the_diagonal <- cbind(0, identity_21_by_20)
on_the_diagonal <- diag(abs(seq.int(-10, 10)))
wilkinson_21 <- below_the_diagonal + above_the_diagonal + on_the_diagonal
 eigen(wilkinson_21)$values
```

- 练习 5-1

  为简单起见，这里我已经手动指定哪些数字是平方。你能想出一个方法来自动确定一个数是否为平方数吗？

```
list(
  "0 to 9" = c(0, 1, 4, 9),
  "10 to 19" = 16,
  "20 to 29" = 25,
  "30 to 39" = 36,
  "40 to 49" = 49,
  "50 to 59" = NULL,
  "60 to 69" = 64,
  "70 to 79" = NULL,
  "80 to 89" = 81,
  "90 to 99" = NULL
)
```

  我们还可以自动计算平方数。以下的 cut 函数将创建不同的范围组：0~9、10~19，依此类推，然后返回一个向量，指出每个平方数在哪一组。然后 split 函数把平方数向量分开成一个列表，其中每个元素包含所对应组里的值：

```
x <- 0:99
sqrt_x <- sqrt(x)
is_square_number <- sqrt_x == floor(sqrt_x)
square_numbers <- x[is_square_number]
groups <- cut(
```

```
  square_numbers,
  seq.int(min(x), max(x), 10),
  include.lowest = TRUE,
  right = FALSE
)
split(square_numbers, groups)
```

- 练习 5-2

  有许多种不同的方法可以获得 iris 数据集的数值子集。实验一下吧！

```
iris_numeric <- iris[, 1:4]
colMeans(iris_numeric)
```

- 练习 5-3

  这个解决方案是许多对数据框取索引和求子集的方法之一：

```
beaver1$id <- 1
beaver2$id <- 2
both_beavers <- rbind(beaver1, beaver2)
subset(both_beavers, as.logical(activ))
```

- 练习 6-1

  我们可以使用 new.env 创建一个环境。之后的语法与列表一样：

```
multiples_of_pi <- new.env()
multiples_of_pi[["two_pi"]] <- 2 * pi
multiples_of_pi$three_pi <- 3 * pi
assign("four_pi", 4 * pi, multiples_of_pi)
ls(multiples_of_pi)
## [1] "four_pi" "three_pi" "two_pi"
```

- 练习 6-2

  这比你想象的要更容易。我们只需要在除以二时使用模运算符来获得余数，然后一切就能自动工作，也包括了非无限值：

```
is_even <- function(x) (x %% 2) == 0
is_even(c(-5:5, Inf, -Inf, NA, NaN))

## [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
## [12]   NA     NA    NA    NA
```

- 练习 6-3

  同样，该函数只需要一行即可。formals 和 body 函数能完成最难的那部分工作，而我们只需要返回其结果列表：

```
args_and_body <- function(fn)
{
  list(arguments = formals(fn), body = body(fn))
}
```

```
args_and_body(var)
```

```
## $arguments
## $arguments$x
##
##
## $arguments$y
## NULL
##
## $arguments$na.rm
## [1] FALSE
##
## $arguments$use
##
##
##

## $body
## {
##     if (missing(use))
##        use <- if (na.rm)
##            "na.or.complete"
##        else "everything"
##     na.method <- pmatch(use, c("all.obs", "complete.obs",
##        "pairwise.complete.obs", "everything", "na.or.complete"))
##     if (is.na(na.method))
##        stop("invalid 'use' argument")
##     if (is.data.frame(x))
##        x <- as.matrix(x)
##     else stopifnot(is.atomic(x))
##     if (is.data.frame(y))
##        y <- as.matrix(y)
##     else stopifnot(is.atomic(y))
##     .Call(C_cov, x, y, na.method, FALSE)
## }
```

```
args_and_body(alarm)
```

```
## $arguments
## NULL
##
## $body
## {
##     cat("\a")
##     flush.console()
## }
```

- 练习 7-1

```
formatC(pi, digits = 16)
```

```
## [1] "3.141592653589793"
```

如果你要把 `formatC` 替换成 `format` 或 `prettyNum` 这也适用。

- 练习 7-2

  调用 strsplit 并使用正则表达式来匹配空格和标点符号。记得 ? 能匹配一个字符：

```
# 使用一个可选的逗号，必须的空格，可以的连字符以及一个可选的空格分开
  strsplit (x, ",? -? ?")

## [[1]]
## [1] "Swan"  "swam"  "over"  "the"  "pond"  "Swim"  "swan"  "swim!"
##
## [[2]]
## [1] "Swan" "swam" "back" "again" "Well" "swum" "swan!"

# 或者把最后的连字符的空格括起来
strsplit(x, ",? (- )?")

## [[1]]
## [1] "Swan"  "swam"  "over"  "the"  "pond"  "Swim"  "swan"  "swim!"
##
## [[2]]
## [1] "Swan"  "swam"  "back"  "again"  "Well"  "swum"  "swan!"
```

- 练习 7-3

  默认情况下，cut 所创建的间隔只包括上限而没有下限。为了获得最小值（3）的计数，我们需要在它下面（例如，在 2）加入一个额外的断点。尝试使用 plyr 包中的 count 替换掉 table：

```
scores <- three_d6(1000)
bonuses <- cut(
  scores,
  c(2, 3, 5, 8, 12, 15, 17, 18),
  labels = -3:3
)
table(bonuses)

## bonuses
##  -3  -2  -1   0   1   2   3
##   4  39 186 486 233  47   5
```

- 练习 8-1

  使用 if 区分其条件。运算符 %in% 能使条件的检查更容易：

```
score <- two_d6(1)
if(score %in% c(2, 3, 12))
{
  game_status <- FALSE
  point <- NA
} else if(score %in% c(7, 11))
{
  game_status <- TRUE
  point <- NA
} else
{
```

```
    game_status <- NA
    point <- score
  }
```

- 练习 8-2

  因为我们不明确代码所要执行的次数，所以使用 `repeat` 循环是最合适的：

```
if(is.na(game_status))
{
  repeat({
    score <- two_d6(1)
    if(score == 7)
    {
      game_status <- FALSE
      break
    } else
    if(score == point)
    {
      game_status <- TRUE
      break
    }
  })
}
```

- 练习 8-3

  因为我们知道需要执行的循环（从最短单词的长度到最长单词的长度）的次数，`for` 循环是最合适的：

```
nchar_sea_shells <- nchar(sea_shells)

for(i in min(nchar_sea_shells):max(nchar_sea_shells))
{
  message("These words have ", i, " letters:")
  print(toString(unique(sea_shells[nchar_sea_shells == i])))
}

## These words have 2 letters:

## [1] "by, So, if, on"

## These words have 3 letters:

## [1] "She, sea, the, The, she, are, I'm"

## These words have 4 letters:

## [1] "sure"

## These words have 5 letters:

## [1] "sells"

## These words have 6 letters:

## [1] "shells, surely"
```

```
## These words have 7 letters:

## [1] ""

## These words have 8 letters:

## [1] "seashore"

## These words have 9 letters:

## [1] "seashells"
```

- 练习 9-1

  由于输入的是一个列表，输出的始终相同而且长度已知，所以 `vapply` 是最好的选择：

```
vapply(wayans, length, integer(1))
```

```
##   Dwayne Kim Keenen Ivory     Damon       Kim     Shawn
##            0            5         4         0         3
##       Marlon        Nadia    Elvira    Diedre    Vonnie
##            2            0         2         5         0
```

- 练习 9-2

  (1) 我们可以通过使用 `str`、`head` 和 `class` 以及阅读帮助页面 `?state.x77` 大致感受一下此数据集：

```
## num [1:50, 1:8] 3615 365 2212 2110 21198 ...
## - attr(*, "dimnames")=List of 2
##  ..$ : chr [1:50] "Alabama" "Alaska" "Arizona" "Arkansas" ...
##  ..$ : chr [1:8] "Population" "Income" "Illiteracy" "Life Exp" ...
```

```
##             Population Income Illiteracy Life Exp Murder HS Grad Frost
## Alabama           3615   3624        2.1    69.05   15.1    41.3    20
## Alaska             365   6315        1.5    69.31   11.3    66.7   152
## Arizona           2212   4530        1.8    70.55    7.8    58.1    15
## Arkansas          2110   3378        1.9    70.66   10.1    39.9    65
## California       21198   5114        1.1    71.71   10.3    62.6    20
## Colorado          2541   4884        0.7    72.06    6.8    63.9   166
##              Area
## Alabama     50708
## Alaska     566432
## Arizona    113417
## Arkansas    51945
## California 156361
## Colorado   103766
```

```
## [1] "matrix"
```

  (2) 现在我们知道此数据是一个矩阵，因此 `apply` 函数最合适。我们要循环每列，因此我们把 2 作为维度值传递给它：

```
## Population    Income  Illiteracy   Life Exp    Murder    HS Grad
##   4246.420  4435.800       1.170     70.879     7.378     53.108
```

```
##      Frost      Area
##    104.460 70735.880

## Population    Income  Illiteracy   Life Exp    Murder    HS Grad
##  4.464e+03 6.145e+02   6.095e-01  1.342e+00 3.692e+00  8.077e+00
##      Frost      Area
##  5.198e+01 8.533e+04
```

- 练习 9-3

  tapply 是 R 基本包中的最佳选择。而 plyr 包中的 ddply 也能很好地工作：

```
with(commute_data, tapply(time, mode, quantile, prob = 0.75))
##  bike   bus   car train
## 63.55 55.62 42.33 36.29

ddply(commute_data, .(mode), summarize, time_p75 = quantile(time, 0.75))
##    mode time_p75
## 1 bike    63.55
## 2  bus    55.62
## 3  car    42.33
## 4 train   36.29
```

- 练习 10-1

  在 R 的图形用户界面中，单击 Package → "Install package(s)... "，选择一个镜像，然后选择 Hmisc。如果下载或安装失败，则请确保你有互联网接入和写入库目录的权限。你也可以尝试访问 CRAN 的网站来手动下载。问题中最常见原因是权限受到限制，在这种情况下请与你的系统管理员联系。

- 练习 10-2

```
install.packages("lubridate")
```

  你可能希望指定一个库进行安装。

- 练习 10-3

  installed.packages 函数能在你的机器上获取软件包的详细信息及其位置（以下的结果可能跟你机器上的位置不同）：

```
pkgs <- installed.packages()
table(pkgs[, "LibPath"])
##
## C:/Program Files/R/R-devel/library                    D:/R/library
##                                  29                             169
```

- 练习 11-1

  此示例使用 strptime 解析和使用 strtime 来进行格式化，除此之外还有其他很多方法：

```
in_string <- c("1940-07-07", "1940-10-09", "1942-06-18", "1943-02-25")
(parsed <- strptime(in_string, "%Y-%m-%d"))
```

```
## [1] "1940-07-07" "1940-10-09" "1942-06-18" "1943-02-25"
```

```
(out_string <- strftime(parsed, "%a %d %b %y"))
```

```
## [1] "Sun 07 Jul 40" "Wed 09 Oct 40" "Thu 18 Jun 42" "Thu 25 Feb 43"
```

- 练习 11-2

  ?Sys.timezone 帮助页面建议了使用以下代码导入时区文件：

  ```
  tzfile <- file.path(R.home("share"), "zoneinfo", "zone.tab")
  tzones <- read.delim(
    tzfile,
    row.names = NULL,
    header = FALSE,
    col.names = c("country", "coords", "name", "comments"),
    as.is = TRUE,
    fill = TRUE,
    comment.char = "#"
  )
  ```

  找到你所在时区的最佳方法取决于你在哪里。从 View(tzones) 开始，如有需要，使用 subset 来缩小搜索范围。

- 练习 11-3

  你可以通过访问一个基础 R 包中的 POSIXlt 日期组件解决此问题，但是使用 lubridate 的 month 和 day 函数会更加清楚：

  ```
  zodiac_sign <- function(x)
  {
    month_x <- month(x, label = TRUE)
    day_x <- day(x)
    switch(
      month_x,
      Jan = if(day_x < 20) "Capricorn" else "Aquarius",
      Feb = if(day_x < 19) "Aquarius" else "Pisces",
      Mar = if(day_x < 21) "Pisces" else "Aries",
      Apr = if(day_x < 20) "Aries" else "Taurus",
      May = if(day_x < 21) "Taurus" else "Gemini",
      Jun = if(day_x < 21) "Gemini" else "Cancer",
      Jul = if(day_x < 23) "Cancer" else "Leo",
      Aug = if(day_x < 23) "Leo" else "Virgo",
      Sep = if(day_x < 23) "Virgo" else "Libra",
      Oct = if(day_x < 23) "Libra" else "Scorpio",
      Nov = if(day_x < 22) "Scorpio" else "Sagittarius",
      Dec = if(day_x < 22) "Sagittarius" else "Capricorn"
    )
  }
  # 可如下使用
  nicolaus_copernicus_birth_date <- as.Date("1473-02-19")
  zodiac_sign(nicolaus_copernicus_birth_date)
  ```

  ```
  ## [1] "Pisces"
  ```

请注意，switch 语句的使用意味着这个函数的实现并非向量化。你可以通过大量使用 ifelse 语句来实现矢量化，或采取更容易地调用 Vectorize(zodiac_sign) 的方法。

- 练习 12-1

  使用 system.file 找到该文件并使用 read.csv 导入数据：

```
hafu_file <- system.file("extdata", "hafu.csv", package = "learningr")
hafu_data <- read.csv(hafu_file)
```

- 练习 12-2

  使用 xlsx 包中的 read.xlsx2 函数导入数据。该数据在第一个（也是唯一的）工作表，而且最好指定每列的数据的类型：

```
library(xlsx)
gonorrhoea_file <- system.file(
  "extdata",
  "multi-drug-resistant gonorrhoea infection.xls",
  package = "learningr"
)
gonorrhoea_data <- read.xlsx2(
  gonorrhoea_file,
  sheetIndex = 1,
  colClasses = c("integer", "character", "character", "numeric")
)
```

- 练习 12-3

  使用 RSQLite 包的方法有好几种。一次一个步骤，我们可以像这样连接数据库：

```
library(RSQLite)
driver <- dbDriver("SQLite")
db_file <- system.file("extdata", "crabtag.sqlite", package = "learningr")
conn <- dbConnect(driver, db_file)

query <- "SELECT * FROM Daylog"
head(daylog <- dbGetQuery(conn, query))

##   Tag ID Mission Day       Date Max Temp Min Temp Max Depth Min Depth
## 1 A03401           0 08/08/2008   27.734   25.203      0.06     -0.07
## 2 A03401           1 09/08/2008   25.203   23.859      0.06     -0.07
## 3 A03401           2 10/08/2008   24.016   23.500     -0.07     -0.10
## 4 A03401           3 11/08/2008   26.453   23.281     -0.04     -0.10
## 5 A03401           4 12/08/2008   27.047   23.609     -0.10     -0.26
## 6 A03401           5 13/08/2008   24.625   23.438     -0.04     -0.13
##   Batt Volts
## 1       3.06
## 2       3.09
## 3       3.09
## 4       3.09
## 5       3.09
## 6       3.09
```

```
dbDisconnect(conn)

## [1] TRUE

dbUnloadDriver(driver)

## [1] TRUE
```

或者放肆一些，使用在第十二章中定义的函数：

```
head(daylog <- query_crab_tag_db("SELECT * FROM Daylog"))
```

或者，我们可以使用 dbReadTable 函数再进一步简化：

```
get_table_from_crab_tag_db <- function(tbl)
{
  driver <- dbDriver("SQLite")
  db_file <- system.file("extdata", "crabtag.sqlite", package = "learningr")
  conn <- dbConnect(driver, db_file)
  on.exit(
    {
      dbDisconnect(conn)
      dbUnloadDriver(driver)
    }
  )
  dbReadTable(conn, tbl)
}
head(daylog <- get_table_from_crab_tag_db("Daylog"))

##   Tag.ID Mission.Day       Date Max.Temp Min.Temp Max.Depth Min.Depth
## 1 A03401           0 08/08/2008   27.734   25.203      0.06     -0.07
## 2 A03401           1 09/08/2008   25.203   23.859      0.06     -0.07
## 3 A03401           2 10/08/2008   24.016   23.500     -0.07     -0.10
## 4 A03401           3 11/08/2008   26.453   23.281     -0.04     -0.10
## 5 A03401           4 12/08/2008   27.047   23.609     -0.10     -0.26
## 6 A03401           5 13/08/2008   24.625   23.438     -0.04     -0.13
##   Batt.Volts
## 1       3.06
## 2       3.09
## 3       3.09
## 4       3.09
## 5       3.09
## 6       3.09
```

- 练习 13-1

  (1) 为了检测问号，使用 str_detect：

```
library(stringr)
data(hafu, package = "learningr")
hafu$FathersNationalityIsUncertain <- str_detect(hafu$Father, fixed("?"))
hafu$MothersNationalityIsUncertain <- str_detect(hafu$Mother, fixed("?"))
```

  (2) 要替换这些问号，使用 str_replace 函数：

```
hafu$Father <- str_replace(hafu$Father, fixed("?"), "")
hafu$Mother <- str_replace(hafu$Mother, fixed("?"), "")
```

- 练习 13-2

  请确保你已经安装了 `reshape2` 包！关键是把测量变量命名为“Father”和“Mother”：

```
hafu_long <- melt(hafu, measure.vars = c("Father", "Mother"))
```

- 练习 13-3

  我们可以使用基础 R 包中的函数来完成，例如 `table` 函数能为我们计数，`sort` 和 `head` 的组合可以使我们能够找到相同的值。把 `useNA = "always"` 传递给 `table` 意味着 `NA` 将始终包含在 `counts` 向量中：

```
top10 <- function(x)
{
  counts <- table(x, useNA = "always")
  head(sort(counts, decreasing = TRUE), 10)
}
top10(hafu$Mother)

## x
##  Japanese      <NA>   English  American    French    German   Russian
##       120        50        29        23        20        12        10
##   Fantasy   Italian Brazilian
##         8         4         3
```

  `plyr` 包提供了另一种替代的解决方案，它能把答案作为一个数据框返回，这对进一步操纵结果可能更为有用：

```
top10_v2 <- function(x)
{
  counts <- count(x)
  head(arrange(counts, desc(freq)), 10)
}
top10_v2(hafu$Mother)

##            x freq
## 1   Japanese  120
## 2       <NA>   50
## 3    English   29
## 4   American   23
## 5     French   20
## 6     German   12
## 7    Russian   10
## 8    Fantasy    8
## 9    Italian    4
## 10 Brazilian    3
```

- 练习 14-1

  (1) `cor` 计算相关性，并且默认为 Pearson 相关性。对其他类型使用 `method` 参数实验一下：

```
with(obama_vs_mccain, cor(Unemployment, Obama))

## [1] 0.2897
```

(2) 以 base/lattice/ggplot2 的顺序，我们有：

```
with(obama_vs_mccain, plot(Unemployment, Obama))
xyplot(Obama ~ Unemployment, obama_vs_mccain)
ggplot(obama_vs_mccain, aes(Unemployment, Obama)) +
  geom_point()
```

- 练习 14-2

(1) 直方图：

```
plot_numbers <- 1:2
layout(matrix(plot_numbers, ncol = 2, byrow = TRUE))
for(drug_use in c(FALSE, TRUE))
{
  group_data <- subset(alpe_d_huez2, DrugUse == drug_use)
  with(group_data, hist(NumericTime))
}

histogram(~ NumericTime | DrugUse, alpe_d_huez2)

ggplot(alpe_d_huez2, aes(NumericTime)) +
  geom_histogram(binwidth = 2) +
  facet_wrap(~ DrugUse)
```

(2) 盒形图：

```
boxplot(NumericTime ~ DrugUse, alpe_d_huez2)
bwplot(DrugUse ~ NumericTime, alpe_d_huez2)

ggplot(alpe_d_huez2, aes(DrugUse, NumericTime, group = DrugUse)) +
geom_boxplot()
```

- 练习 14-3

为简单起见，我们只使用 ggplot2 给出此答案。由于这是一个数据探索，所以没有“正确”的答案：如果图中显示了一些你感到有趣的事情，那么这是值得画出的。当你有几个“干扰因素”（在本例中，我们有年龄 / 种族 / 性别）存在时，对于不同的绘图会有很多不同的可能性。一般情况下，处理多变量有两种策略：先作一个总体概述再添加变量；或者一开始就把所有包含在内，然后再删除那些看起来不那么有趣的变量。我们将使用第二种策略。

要看到每个变量的影响，最简单的方法的是把所有的东西都放到面板上：

```
ggplot(gonorrhoea, aes(Age.Group, Rate)) +
  geom_bar(stat = "identity") +
  facet_wrap(~ Year + Ethnicity + Gender)
```

每个面板上的条形高度是不同的，特别是种族，但是我们很难在 50 个面板中看出是怎么回事。我们可以把每年的数据都画在相同的面板来进行简化。我们使用 group 来声明哪些值属于同一栏，使用 fill 为每个条形指定不同的填充颜色，还可使用 position = "dodge" 把条形彼此分开，而不是堆叠在一起：

```
ggplot(gonorrhoea, aes(Age.Group, Rate, group = Year, fill = Year)) +
  geom_bar(stat = "identity", position = "dodge") +
  facet_wrap(~ Ethnicity + Gender)
```

面板数量减少是好事，但我还是觉得很难从那些条形中获取很多信息。由于大部分年龄组宽度的（五年）相同，我们可以稍用点手段：画一条以年龄为 x 轴的线图。虽然因为年龄组更宽一些，该图在每个面板的右侧看上去有点不对劲，但它已经可以提供某些信息了：

```
ggplot(gonorrhoea, aes(Age.Group, Rate, group = Year, color = Year)) +
  geom_line() +
  facet_wrap(~ Ethnicity + Gender)
```

该线紧靠在一起，所以看上去似乎并没有显示出太大的时间趋势（虽然时间段正好是五年）。由于有两种切面的方式，我们可以通过使用 facet_grip 而不是 facet_wrap 稍微改善一下绘图：

```
ggplot(gonorrhoea, aes(Age.Group, Rate, group = Year, color = Year)) +
  geom_line() +
  facet_grid(Ethnicity ~ Gender)
```

这清楚地表明种族对淋病感染率的影响：曲线要比在“非西班牙裔黑人”和“美洲印第安人和阿拉斯加土著”的群体高得多。由于这些群体占据了大部分，所以很难看到性别的影响。通过给每一行分配一个不同的 y 轴，它可以中和种族的影响，并更加清楚地看到男性和女性之间的区别：

```
ggplot(gonorrhoea, aes(Age.Group, Rate, group = Year, color = Year)) +
  geom_line() +
  facet_grid(Ethnicity ~ Gender, scales = "free_y")
```

在这里你可以看到女性的感染率高于男性（在相同年龄组和民族中），并有一个更加持续的高峰期：从 15 到 24 岁；而男性为 20 到 24 岁。

- 练习 15-1

  (1) 在这种情况下，无论是错别字的数目 x 和错别字的平均数目 lambda 都是 3：

```
dpois(3, 3)

## [1] 0.224
```

  (2) 位数 q 为 12（个月），size 为 1，因为我们只需要怀孕一次，每个月的 probability 为 0.25：

```
pnbinom(12 , 1, 0.25 )
```

```
## [1] 0.9762
```

(3) 为了更加直截了当，我们只把 `probability` 设定为 `0.9`：

```
Qbirthday(0.9)
```

```
## [1] 41
```

- 练习 15-2

  我们要么可以取数据集的一个子集（使用通常的索引方法或 `subset` 函数），要么把子集的详细信息传递给 `lm` 的 `subset` 参数：

```
model1 <- lm(
  Rate ~ Year + Age.Group + Ethnicity + Gender,
  gonorrhoea,
  subset = Age.Group %in% c("15 to 19" ,"20 to 24" ,"25 to 29" ,"30 to 34")
)
summary(model1)
```

```
##
## Call:
## lm(formula = Rate ~ Year + Age.Group + Ethnicity + Gender, data = gonorrhoea,
##     subset = Age.Group %in% c("15 to 19", "20 to 24", "25 to 29",
##         "30 to 34"))
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -774.0 -127.7  -10.3  106.2  857.7
##
## Coefficients:
##                                 Estimate Std. Error t value Pr(>|t|)
## (Intercept)                      9491.13   25191.00    0.38   0.7068
## Year                               -4.55      12.54   -0.36   0.7173
## Age.Group20 to 24                 131.12      50.16    2.61   0.0097
## Age.Group25 to 29                -124.60      50.16   -2.48   0.0138
## Age.Group30 to 34                -259.83      50.16   -5.18  5.6e-07
## EthnicityAsians & Pacific Islanders -212.76    56.08   -3.79   0.0002
## EthnicityHispanics               -124.06      56.08   -2.21   0.0281
## EthnicityNon-Hispanic Blacks     1014.35      56.08   18.09  < 2e-16
## EthnicityNon-Hispanic Whites     -174.72      56.08   -3.12   0.0021
## GenderMale                        -83.85      35.47   -2.36   0.0191
##
## (Intercept)
## Year
## Age.Group20 to 24                **
## Age.Group25 to 29                *
## Age.Group30 to 34                ***
## EthnicityAsians & Pacific Islanders ***
## EthnicityHispanics               *
## EthnicityNon-Hispanic Blacks     ***
## EthnicityNon-Hispanic Whites     **
```

```
## GenderMale                       *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 251 on 190 degrees of freedom
## Multiple R-squared:  0.798, Adjusted R-squared:  0.789
## F-statistic: 83.7 on 9 and 190 DF,  p-value: <2e-16
```

Year 不显著，因此我们将其删除：

```
model2 <- update(model1, ~ . - Year)
summary(model2)

##
## Call:
## lm(formula = Rate ~ Age.Group + Ethnicity + Gender, data = gonorrhoea,
##     subset = Age.Group %in% c("15 to 19", "20 to 24", "25 to 29",
##         "30 to 34"))
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -774.0 -129.3   -6.7  104.3  866.8
##
## Coefficients:
##                                  Estimate Std. Error t value Pr(>|t|)
## (Intercept)                         358.2       53.1    6.75  1.7e-10
## Age.Group20 to 24                   131.1       50.0    2.62  0.00949
## Age.Group25 to 29                  -124.6       50.0   -2.49  0.01363
## Age.Group30 to 34                  -259.8       50.0   -5.19  5.3e-07
## EthnicityAsians & Pacific Islanders   -212.8       55.9   -3.80  0.00019
## EthnicityHispanics                 -124.1       55.9   -2.22  0.02777
## EthnicityNon-Hispanic Blacks       1014.3       55.9   18.13  < 2e-16
## EthnicityNon-Hispanic Whites       -174.7       55.9   -3.12  0.00207
## GenderMale                          -83.8       35.4   -2.37  0.01881
##
## (Intercept)                      ***
## Age.Group20 to 24                **
## Age.Group25 to 29                *
## Age.Group30 to 34                ***
## EthnicityAsians & Pacific Islanders ***
## EthnicityHispanics               *
## EthnicityNon-Hispanic Blacks     ***
## EthnicityNon-Hispanic Whites     **
## GenderMale                       *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 250 on 191 degrees of freedom
## Multiple R-squared:  0.798, Adjusted R-squared:  0.79
## F-statistic: 94.5 on 8 and 191 DF,  p-value: <2e-16
```

这一次，Gender 就变得显著了，因此我们也可以停止了。（儿童和老人的感染率在性别上都很接近，因为性对于它们来说并非传播的主要原因，但是对于成年女性来说，她们的感染率要高于男性。）

大多数可能的交互项都不是非常有趣。通过增加一个种族和性别的交互，我们可以得到了一个更好的拟合，虽然不是所有的交互项都是显著的（输出非常详细，所以未显示）。如果你仍然对此分析感兴趣，尝试为黑色 / 非黑人创建一个单独的民族指示器，然后使用它作为与性别的交互项：

```
##
## Call:
## lm(formula = Rate ~ Age.Group + Ethnicity + Gender + Ethnicity:Gender,
##     data = gonorrhoea, subset = Age.Group %in% c("15 to 19",
##         "20 to 24", "25 to 29", "30 to 34"))
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -806.0 -119.4   -9.4  105.3  834.8
##
## Coefficients:
##                                              Estimate Std. Error t value
## (Intercept)                                     405.1       63.9    6.34
## Age.Group20 to 24                               131.1       50.1    2.62
## Age.Group25 to 29                              -124.6       50.1   -2.49
## Age.Group30 to 34                              -259.8       50.1   -5.19
## EthnicityAsians & Pacific Islanders           -296.9       79.2   -3.75
## EthnicityHispanics                             -197.2       79.2   -2.49
## EthnicityNon-Hispanic Blacks                    999.5       79.2   12.62
## EthnicityNon-Hispanic Whites                   -237.0       79.2   -2.99
## GenderMale                                     -177.6       79.2   -2.24
## EthnicityAsians & Pacific Islanders:GenderMale    168.2      112.0    1.50
## EthnicityHispanics:GenderMale                   146.2      112.0    1.30
## EthnicityNon-Hispanic Blacks:GenderMale          29.7      112.0    0.27
## EthnicityNon-Hispanic Whites:GenderMale         124.6      112.0    1.11
##                                              Pr(>|t|)
## (Intercept)                                   1.7e-09 ***
## Age.Group20 to 24                             0.00960 **
## Age.Group25 to 29                             0.01377 *
## Age.Group30 to 34                             5.6e-07 ***
## EthnicityAsians & Pacific Islanders          0.00024 ***
## EthnicityHispanics                            0.01370 *
## EthnicityNon-Hispanic Blacks                  < 2e-16 ***
## EthnicityNon-Hispanic Whites                  0.00315 **
## GenderMale                                    0.02615 *
## EthnicityAsians & Pacific Islanders:GenderMale  0.13487
## EthnicityHispanics:GenderMale                 0.19361
## EthnicityNon-Hispanic Blacks:GenderMale       0.79092
## EthnicityNon-Hispanic Whites:GenderMale       0.26754
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 251 on 187 degrees of freedom
## Multiple R-squared:  0.802, Adjusted R-squared:  0.789
## F-statistic: 63.2 on 12 and 187 DF,  p-value: <2e-16
```

- 练习 15-3

  首先解决安装问题。像通常一样获取软件包并安装：

```
install.packages("betareg")
```

我们需要重新调整响应变量为 0 和 1 之间（而不是 0 和 100），因为这是 β 分布的范围：

```
ovm <- within(obama_vs_mccain, Obama <- Obama / 100)
```

现在我们准备运行模型。它与运行一元线性回归相同，但我们称之为 betareg 而不是 lm。我随机地关注种族群体 Black 和宗教 Protestant：

```
library(betareg)
beta_model1 <- betareg(
 Obama ~ Turnout + Population + Unemployment + Urbanization + Black + Protestant,
 ovm,
 subset = State != "District of Columbia"
)
summary(beta_model1)

##
## Call:
## betareg(formula = Obama ~ Turnout + Population + Unemployment +
##     Urbanization + Black + Protestant, data = ovm, subset = State !=
##     "District of Columbia")
##
## Standardized weighted residuals 2:
##    Min     1Q Median     3Q    Max
## -2.834 -0.457 -0.062  0.771  2.044
##
## Coefficients (mean model with logit link):
##               Estimate Std. Error z value Pr(>|z|)
## (Intercept)  -5.52e-01   4.99e-01   -1.11   0.2689
## Turnout       1.69e-02   5.91e-03    2.86   0.0042 **
## Population    1.66e-09   5.34e-09    0.31   0.7566
## Unemployment  5.74e-02   2.31e-02    2.48   0.0132 *
## Urbanization -1.82e-05   1.54e-04   -0.12   0.9059
## Black         8.18e-03   4.27e-03    1.91   0.0558 .
## Protestant   -1.78e-02   3.17e-03   -5.62  1.9e-08 ***
##
## Phi coefficients (precision model with identity link):
##       Estimate Std. Error z value Pr(>|z|)
## (phi)    109.5       23.2    4.71  2.5e-06 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Type of estimator: ML (maximum likelihood)
## Log-likelihood: 72.1 on 8 Df
## Pseudo R-squared: 0.723
## Number of iterations: 36 (BFGS) + 3 (Fisher scoring)
```

Urbanization 是不显著的，所以我们将其删除，使用与 lm 相同的技术：

```
beta_model2 <- update(beta_model1, ~ . - Urbanization)
summary(beta_model2)
```

```
##
## Call:
## betareg(formula = Obama ~ Turnout + Population + Unemployment +
##     Black + Protestant, data = ovm, subset = State != "District of Columbia")
##
## Standardized weighted residuals 2:
##    Min     1Q Median     3Q    Max
## -2.831 -0.457 -0.053  0.774  2.007
##
## Coefficients (mean model with logit link):
##               Estimate Std. Error z value Pr(>|z|)
## (Intercept)  -5.69e-01   4.77e-01   -1.19   0.2327
## Turnout       1.70e-02   5.86e-03    2.90   0.0037 **
## Population    1.73e-09   5.31e-09    0.32   0.7452
## Unemployment  5.70e-02   2.29e-02    2.48   0.0130 *
## Black         7.93e-03   3.73e-03    2.13   0.0334 *
## Protestant   -1.76e-02   2.48e-03   -7.09  1.3e-12 ***
##
## Phi coefficients (precision model with identity link):
##       Estimate Std. Error z value Pr(>|z|)
## (phi)    109.4       23.2    4.71  2.5e-06 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Type of estimator: ML (maximum likelihood)
## Log-likelihood: 72.1 on 7 Df
## Pseudo R-squared: 0.723
## Number of iterations: 31 (BFGS) + 3 (Fisher scoring)
```

同样，也要把 Population 去掉：

```
beta_model3 <- update(beta_model2, ~ . - Population)
summary(beta_model3)
```

```
##
## Call:
## betareg(formula = Obama ~ Turnout + Unemployment + Black + Protestant,
##     data = ovm, subset = State != "District of Columbia")
##
## Standardized weighted residuals 2:
##    Min     1Q Median     3Q    Max
## -2.828 -0.458  0.043  0.742  1.935
##
## Coefficients (mean model with logit link):
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept)  -0.55577    0.47567   -1.17   0.2427
## Turnout       0.01686    0.00585    2.88   0.0040 **
## Unemployment  0.05964    0.02145    2.78   0.0054 **
## Black         0.00820    0.00364    2.26   0.0241 *
## Protestant   -0.01779    0.00240   -7.42  1.2e-13 ***
##
## Phi coefficients (precision model with identity link):
##       Estimate Std. Error z value Pr(>|z|)
## (phi)    109.2       23.2    4.71  2.5e-06 ***
```

```
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Type of estimator: ML (maximum likelihood)
## Log-likelihood: 72.1 on 6 Df
## Pseudo R-squared: 0.723
## Number of iterations: 27 (BFGS) + 2 (Fisher scoring)
```

绘图可以使用与 lm 完全相同的代码来完成：

```
plot_numbers <- 1:6
layout(matrix(plot_numbers, ncol = 2, byrow = TRUE))
plot(beta_model3, plot_numbers)
```

- 练习 16-1

  对于检查和处理用户输入，并没有须特别遵循的规则。在一般情况下，如果输入的形式是错误的，最好是设法将其转换成正确的，并在此过程中警告用户。因此，对于非数字输入，我们可以尝试将它们转换为数值，并抛出一个警告。

  对于非正值，我们可以尝试替换一个范围内的值，或假装该值是缺失的，但这样会使答案出错。在这种情况下，最好是抛出一个错误：

```
harmonic_mean <- function(x, na.rm = FALSE)
{
  if(!is.numeric(x))
  {
    warning("Coercing 'x' to be numeric.")
    x <- as.numeric(x)
  }
  if(any(!is.na(x) & x <= 0))
  {
    stop("'x' contains non-positive values")
  }
  1 / mean(1 / x, na.rm = na.rm)
}
```

- 练习 16-2

  这里是 RUnit 的版本。为了测试缺失值，我们需要考虑的情况有：na.rm 是 TRUE 和 na.rm 为 FALSE。为了测试须抛出警告的情况，我们略施小计，将 warn 选项改为 2，意即将警告看作错误。为了测试非正值，使用边界数值零。

```
test.harmonic_mean.1_2_4.returns_12_over_7 <- function()
{
  expected <- 12 / 7
  actual <- harmonic_mean(c(1, 2, 4))
  checkEqualsNumeric(expected, actual)
}
test.harmonic_mean.no_inputs.throws_error <- function()
{
  checkException(harmonic_mean())
```

```
}
test.harmonic_mean.some_missing.returns_na <- function()
{
  expected <- NA_real_
  actual <- harmonic_mean(c(1, 2, 4, NA))
  checkEqualsNumeric(expected, actual)
}
test.harmonic_mean.some_missing_with_nas_removed.returns_12_over_7 <- function()
{
  expected <- 12 / 7
  actual <- harmonic_mean(c(1, 2, 4, NA), na.rm = TRUE)
  checkEqualsNumeric(expected, actual)
}
test.harmonic_mean.non_numeric_input.throws_warning <- function()
{
  old_ops <- options(warn = 2)
  on.exit(options(old_ops))
  checkException(harmonic_mean("1"))
}
test.harmonic_mean.zero_inputs.throws_error <- function()
{
  checkException(harmonic_mean(0))
}
```

翻译成 testthat 的版本很简单。使用 expect_warning 函数，可使警告更容易测试：

```
expect_equal(12 /7, harmonic_mean(c(1, 2, 4)))
expect_error(harmonic_mean())
expect_equal(NA_real_, harmonic_mean(c(1, 2, 4, NA)))
expect_equal(12 /7, harmonic_mean(c(1, 2, 4, NA), na.rm = TRUE))
expect_warning(harmonic_mean("1"))
expect_error(harmonic_mean(0))
```

- 练习 16-3

  下面是更新的 harmonic_mean 函数：

```
harmonic_mean <- function(x, na.rm = FALSE)
{
  if(!is.numeric(x))
  {
    warning("Coercing 'x' to be numeric.")
    x <- as.numeric(x)
  }
  if(any(!is.na(x) & x <= 0))
  {
    stop("'x' contains non-positive values")
  }
  result <- 1 / mean(1 / x, na.rm = na.rm)
  class(result) <- "harmonic"
  result
}
```

为了生成一个 S3 的方法，名称的格式必须符合 function.class 的形式，在本例中即为

print.harmonic。其他内容可以与其他 print 函数生成，但在这里，我们使用较低级别的 cat 函数：

```
print.harmonic <- function(x, ...)
{
  cat("The harmonic mean is", x, "\n", ...)
}
```

- 练习 17-1

  创建函数和数据框都很简单：

```
sum_of_squares <- function(n)
{
  n * (n + 1) * (2 * n + 1) / 6
}
x <- 1:10
squares_data <- data.frame(x = x, y = sum_of_squares(x))
```

  在你调用 package.skeleton 时，需要考虑在哪个目录下创建软件包：

```
package.skeleton("squares", c("sum_of_squares", "squares_data"))
```

- 练习 17-2

  此函数的文档应该置于 R/sum_of_squares.R，或类似的：

```
#' Sum of Squares
#'
#' Calculates the sum of squares of the first \code{n} natural numbers.
#'
#' @param n A positive integer.
#' @return An integer equal to the sum of the squares of the first \code{n}
#' natural numbers.
#' @author Richie Cotton.
#' @seealso \code{\link[base]{sum}}
#' @examples
#' sum_of_squares(1:20)
#' @keywords misc
#' @export
```

  该软件包的文档位于 R/squares-package.R 中：

```
#' squares: Sums of squares.
#'
#' A test package that contains data on the sum of squares of natural
#' numbers, and a function to calculate more values.
#'
#' @author Richie Cotton
#' @docType package
#' @name squares
#' @aliases squares squares-package
#' @keywords package
NULL
```

数据文档可以被放入同一个文件，或是在 R/squares-data.R 中：

```
#' Sum of squares dataset
#'
#' The sum of squares of natural numbers.
#' \itemize{
#' \item{x}{Natural numbers.}
#' \item{y}{The sum of squares from 1 to \code{x}.}
#' }
#'
#' @docType data
#' @keywords datasets
#' @name squares_data
#' @usage data(squares_data)
#' @format A data frame with 10 rows and 2 variables.
NULL
```

- 练习 17-3

  代码很容易，困难的部分在于修复任何问题：

```
check("squares")
build("squares")
```

参考文献

1. Jason R. Briggs. *Python for Kids: A Playful Introduction to Programming*. 2012. William Pollock. ISBN-13 978-1-59327-407-8.

2. Garrett Grolemund. *Data Analysis with R*. 2013. O'Reilly. ISBN-13 978-1-4493-5901-0.

3. Andrie de Vries and Joris Meys.*R For Dummies*. 2012. John Wiley & Sons. ISBN-13 978-1-1199-6284-7.

4. Michael Fitzgerald. *Introducing Regular Expressions*. 2012. O'Reilly. ISBN-13 978-1-4493-9268-0.

5. Paul Murrell. *R Graphics*, Second Edition. 2011. Chapman and Hall/CRC. ISBN-13 978-1-4398-3176-2.

6. Hadley Wickham. *ggplot2: Elegant Graphics for Data Analysis*. 2010. Springer. ISBN-13 978-0-3879-8140-6.

7. Deepayan Sarkar.*Lattice: Multivariate Data Visualization with R*. 2008. Springer. ISBN-13 978-0-3877-5968-5.

8. Edward R. Tufte. *Envisioning Information*. 1990. Graphics Press USA. ISBN-13 978-0-9613-9211-6.

9. Michael J. Crawley. *The R Book*. 2013. John Wiley & Sons. ISBN-13 978-0-4709-7392-9.

10. Andy Field, Jeremy Miles, and Zoe Field. *Discovering Statistics Using R*. 2012. SAGE Publications. ISBN-13 978-1-4462-0046-9.

11. Max Kuhn. *Applied Predictive Modeling*. 2013. Springer. ISBN-13 978-1-4614-6848-6.

12. John Fox and Sanford Weisberg. *An R Companion to Applied Regression*. 2011. SAGE Publications. ISBN-13 978-1-4129-7514-8.

13. José Pinheiro and Douglas Bates. *Mixed-Effects Models in S and S-PLUS*. 2009. Springer. ISBN-13 978-1-4419-0317-4.

14. Graham Williams. *Data Mining with Rattle and R: The Art of Excavating Data for Knowledge Discovery*. 2011. Springer. ISBN-13 978-1-4419-9889-7.

15. Thomas Lumley. "Standard nonstandard evaluation rules". 2003. *http://developer.r-project.org/nonstandard-eval.pdf*.

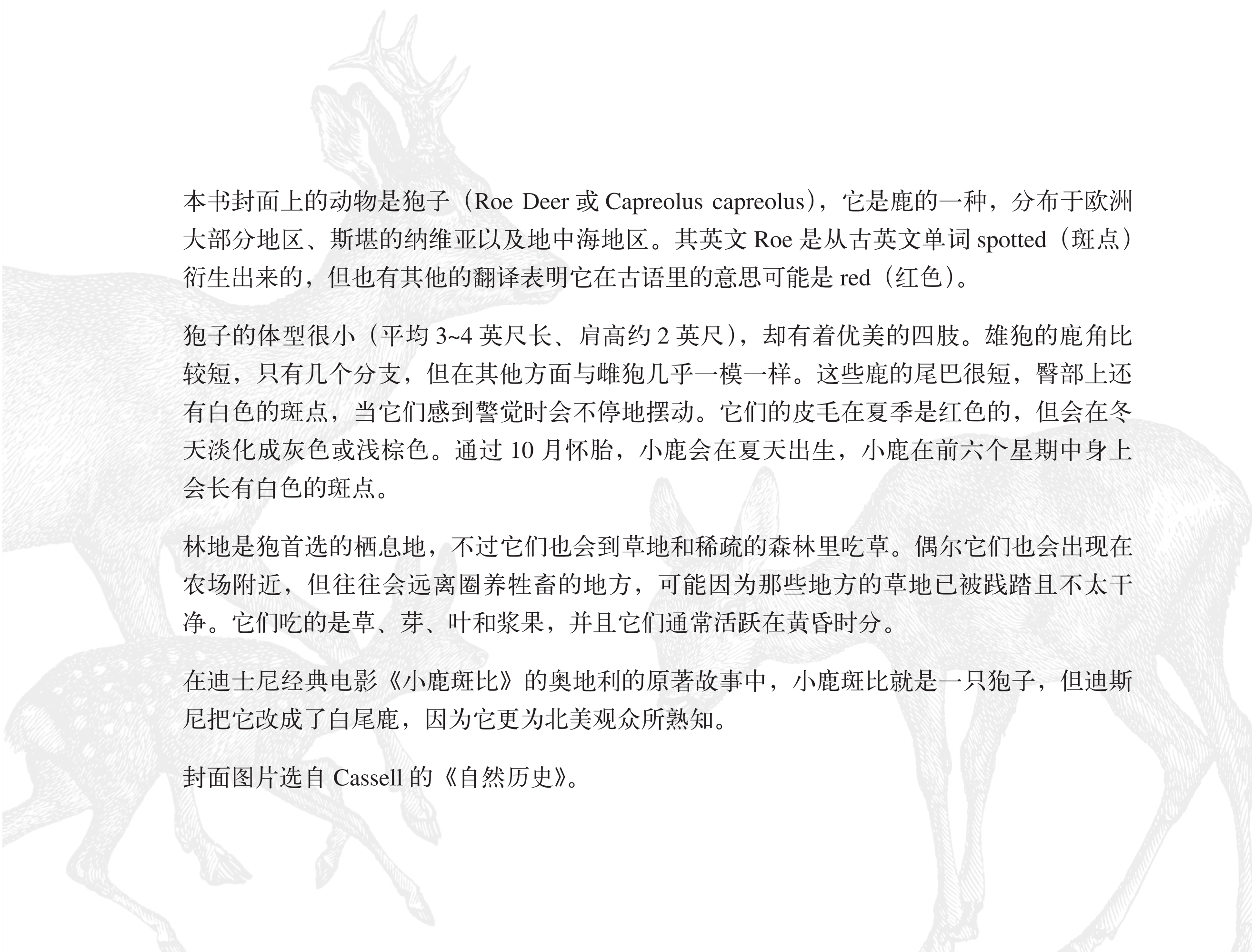
16. Dirk Eddelbuettel. *Seamless R and C++ Integration with Rcpp*. 2013. Springer. ISBN-13 978-1-4614-6867-7.

17. Yihui Xie. *Dynamic Documents with R and knitr*. 2013. Chapman and Hall/CRC. ISBN-13 978-1-4822-0353-0.

18. Michael Lawrence and John Verzani. *Programming Graphical User Interfaces in R*. 2012. Chapman and Hall/CRC. ISBN-13 978-1-4398-5682-6.

19. Q. Ethan McCallum and Stephen Weston. *Parallel R*. 2012. O'Reilly. ISBN-13 978-1-4493-0992-3.

关于封面



本书封面上的动物是狍子（Roe Deer 或 Capreolus capreolus），它是鹿的一种，分布于欧洲大部分地区、斯堪的纳维亚以及地中海地区。其英文 Roe 是从古英文单词 spotted（斑点）衍生出来的，但也有其他的翻译表明它在古语里的意思可能是 red（红色）。

狍子的体型很小（平均 3~4 英尺长、肩高约 2 英尺），却有着优美的四肢。雄狍的鹿角比较短，只有几个分支，但在其他方面与雌狍几乎一模一样。这些鹿的尾巴很短，臀部上还有白色的斑点，当它们感到警觉时会不停地摆动。它们的皮毛在夏季是红色的，但会在冬天淡化成灰色或浅棕色。通过 10 月怀胎，小鹿会在夏天出生，小鹿在前六个星期中身上会长有白色的斑点。

林地是狍首选的栖息地，不过它们也会到草地和稀疏的森林里吃草。偶尔它们也会出现在农场附近，但往往会远离圈养牲畜的地方，可能因为那些地方的草地已被践踏且不太干净。它们吃的是草、芽、叶和浆果，并且它们通常活跃在黄昏时分。

在迪士尼经典电影《小鹿斑比》的奥地利的原著故事中，小鹿斑比就是一只狍子，但迪斯尼把它改成了白尾鹿，因为它更为北美观众所熟知。

封面图片选自 Cassell 的《自然历史》。

学习R

本书讲解如何使用R语言及其软件环境分析数据，即使没有编程经验也能看懂。通过这本实用教程，你可以轻松掌握如何使用必要的R工具来分析数据，同时掌握相关数据类型和通用的编程概念。

本书后半部分会讲到数据分析的各种实际应用，涵盖导入数据和发布结果。另外，值得一提的是，本书每一章都会结合所讲内容提供精心编制的小测试和练习题，需要编写R代码完成，从而巩固所学的知识。

- 编写简单的R程序，知道R语言能做什么
- 使用向量、数组、列表、数据框和字符串等数据类型
- 掌握条件语句以及分支和循环控制语句
- 应用R的扩展包，将你自己的工作成果打包发给其他人
- 清理从各种来源导入的数据
- 通过可视化和汇总统计理解数据
- 使用统计模型传递关于数据的定量判断并进行预测
- 了解编写数据分析代码时出现错误的应对措施

Richard Cotton是一位通晓化学安全及健康的数据科学家，开发过很多能让非专业用户访问统计模型的工具。他开发了很多R包，如assertive（用于检查变量的状态）和sig（用于确保功能具有合理的API）。他也是The Damned Liars公司的统计学顾问。

“拥挤不堪的R图书一族迎来了一位清新、不落俗套、风趣幽默的新成员。阅读本书是十足的享受，本书令我大开眼界。”

——John Verzani
纽约州立大学/斯坦顿大学

“这本书读起来感觉棒极了，示例完整清晰，内容通俗易懂，是目前市面上介绍R核心组件最优秀的图书之一。”

——Rebecca Smith
TDX公司分析经理

“本书正是你梦寐以求的R图书！不要只看封面，直接翻开书看内容吧。”

——JD Long
CerebralMastication.com
资深博主

封面设计：Karen Montgomery，张健

O'REILLY®
oreilly.com.cn

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机/数据分析

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-115-35170-8

ISBN 978-7-115-35170-8
定价：69.00元

看完了